creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학박사 학위논문

Applying Reinforcement Learning to Mitigate Long Tail Latency Problem in SSD

SSD의 긴 꼬리 지연시간 문제 완화를 위한 강화학습의 적용

2020 년 2 월

서울대학교 대학원

컴퓨터공학부

강 원 경

Applying Reinforcement Learning to Mitigate Long Tail Latency Problem in SSD

SSD의 긴 꼬리 지연시간 문제 완화를 위한
강화학습의 적용

지도교수 유 승 주

이 논문을 공학박사 학위논문으로 제출함

2019 년 11 월

서울대학교 대학원

컴퓨터공학부

강 원 경

강원경의 공학박사 학위논문을 인준함

2019 년 12 월

위 원 장 이 재 욱 (인)

부위원장 유 승 주 (인)

위 원 김 진 수 (인)

위 원 신 동 군 (인)

위 원 윤 성 로 (인)

Abstract

Applying Reinforcement Learning to Mitigate Long Tail Latency Problem in SSD

Won Kyung Kang

Department of Computer Science and Engineering

The Graduate School

Seoul National University

NAND flash memory is widely used in a variety of systems, from real-time embedded systems to high-performance enterprise server systems. Flash memory has (1) erase-before-write (write-once) and (2) endurance problems. To handle the erase-before-write feature, apply a flash-translation layer (FTL). Currently, the page-level mapping method is mainly used to reduce the latency increase caused by the write-once and block erase characteristics of flash memory.

Garbage collection (GC) is one of the leading causes of long-tail latency, which increases more than 100 times the average latency at 99^{th} percentile. Therefore, real-time systems or quality-critical systems cannot satisfy given requirements such as QoS restrictions.

As flash memory capacity increases, GC latency also tends to increase. This is because the block size (the number of pages included in one block) of the flash memory increases as the capacity of the flash memory increases. GC latency is determined by valid page copy and block erase time. Therefore, as block size increases, GC latency also increases.

Especially, the block size gets increased from 2D to 3D NAND flash memory, e.g., 256 pages/block in 2D planner NAND flash memory and 768 pages/block in 3D NAND flash memory. Even in 3D NAND flash memory, the block size is expected to continue to increase. Thus, the long write latency problem incurred by GC can become more serious in 3D NAND flash memory-based storage.

In this dissertation, we propose three versions of the novel GC scheduling method based on reinforcement learning. The purpose of this method is to reduce the long tail latency caused by GC by utilizing the idle time of the storage system. Also, we perform a quantitative analysis for the RL-assisted GC solution.

RL-assisted GC scheduling technique was proposed which learns the storage access behavior online and determines the number of GC operations to exploit the idle time. We also presented aggressive methods,which helps in further reducing the long tail latency by aggressively performing fine-grained GC operations.

We also proposed a technique that dynamically manages key states in

RL-assisted GC to reduce the long-tail latency. This technique uses many fine-grained pieces of information as state candidates and manages key states that suitably represent the characteristics of the workload using a relatively small amount of memory resource. Thus, the proposed method can reduce the long-tail latency even further.

In addition, we presented a Q-value prediction network that predicts the initial Q-value of a newly inserted state in the Q-table cache. The integrated solution of the Q-table cache and Q-value prediction network can exploit the short-term history of the system with a low-cost Q-table cache. It is also equipped with a small network called Q-value prediction network to make use of the long-term history and provide good Q-value initialization for the Q-table cache. The experiments show that our proposed method reduces by 25%-37% the long tail latency compared to the state-of-the-art method.

Keywords: Solid state drive, long tail latency, garbage collection, reinforcement learning, storage, nand flash memory

Student Number: 2016-30281

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Flash memory storages are widely used in embedded systems, and consumer and enterprise-server systems. Flash memory has two principal issues: (1) erase-before-write (write-once) property, and (2) endurance problem. To address the erase-before-write property, a flash translation layer (FTL) is employed. Currently, a page-level mapping [3] is being widely used to reduce the write latency induced by write-once and bulk-erase properties of flash memory storages. In the page-level mapping, when writing new data, FTL assigns a new free page, and subsequently, writes data to the newly assigned free page. Thereafter, it updates the address-mapping information between the logical and the physical addresses. If the free blocks are insufficient, they are obtained by reclaiming the unused space in the used blocks. To do that, the valid pages of the victim block are copied to a new block. The victim block is then erased to obtain a free block. This procedure is called garbage collection (GC). The GC induces a long-latency problem because the page-copy and block-erase operations are time-consuming.

GC latency increases as the capacity of flash memory increases. It is mainly due to the fact that the block size (number of pages per block) increases as the capacity of flash memory increases. GC latency is determined by the time for valid page copy and block erase. Thus, as block size gets increased, GC latency also increases. According to our analysis, the block size has a strong impact on long tail latency. Especially, the block size gets increased from 2D to 3D NAND flash memory, e.g., 256 pages/block in 2D planner NAND flash memory [4] and 768 pages/block in 3D NAND flash memory [5]. Even in 3D NAND flash memory, the block size is expected to continue to increase [6, 7]. Thus, the long write latency problem incurred by GC can become more serious in 3D NAND flash memory-based storage. Note that the long write latency due to GC can increase not only write latency but also read latency since GC can stall the service of subsequent read requests. A long tail is observed in the distribution of the write latency because of the GC. For instance, the latency at the 99^{th} percentile can be $100\times$ higher than the average latency [8]. Such a long-tail latency causes a significant problem in real-time embedded and enterprise-server systems which need to meet the real-time and quality of service (QoS) requirements.

In this dissertation, we propose a reinforcement learning-assisted GC technique to reduce the long tail latency. The proposed technique is a new approach to exploit the idle time in the storage with reinforcement learning. In addition, in order to reduce the memory cost of large Q-table

management while benefiting from a large number of states, we propose a method called *Q-table cache* which aims to store recently visited states among a large number of state candidates and further reduces long tail latency by efficiently exploiting the large number of states. We also report that the Q-table cache alone has a limitation in initializing new entries to the cache structure. It is because an insertion of a new entry to the Q-table cache requires an appropriate initialization of its Q-value. However, a naive solution of zero initialization proves limited in fully exploiting the potential of the Q-table cache. In order to resolve this problem, we propose a novel notion of Q-value prediction and a neural network called Q-value prediction network (QP Net) as a realization.

The contributions of this study are as follows:

- The proposed reinforcement learning-assisted solution helps determine the number of GC operations to be executed to exploit the varying idle time while avoiding the long-tail latency due to the GC [9].

- We also present an optimization scheme that aggressively performs fine-grained GC to prepare free blocks in advance, thereby reducing the blockage due to the GC, which significantly reduces the long-tail latency [9].

- We propose a technique called Q-table cache that requires only a small amount of memory space to represent the environment of the

RL model by managing key states dynamically [10].

- This study offers the QP Net for further reduction in long tail latency by addressing the problem of zero Q-value initialization in the original Q-table cache. The proposed QP Net learns the system behavior during runtime thereby being able to provide good initial Q-values in case of inserting a new entry to the Q-table cache. This improves Q-learning on the Q-table cache while finally contributing to further reduction in long-tail latency. Thus, our integrated solution of Q-table cache and QP Net aims at exploiting both short-term (by Q-table cache) and long-term (by QP Net) history of system behavior [11].

- We perform in-depth analysis of storage workloads and identify storage-intensive ones. We show that our new scheme offers significant reductions in long-tail latency for those workloads compared with the state-of-the-art technique [11].

The rest of this dissertation is organized as follows: Chapter 2 describes the background of the long-tail latency, flash storage system and RL. Chapter 3 reviews previous GC and RL techniques. Chapter 4 explains small Q-table based solution to reduce long-tail latency problems. Chapter 5 presents Q-table cache based solution to exploit a large number of states at small cost. Chapter 6 proposes a neural network-based solution combining Q-table cache to exploit both long and short-term

histories, and Chapter 7 concludes the dissertation.

Chapter 2

Background

2.1 System Level Tail Latency

Modern applications require small and predictable response times. The response behaviors are characterized by strict performance service level objective (SLOs), e.g., 99.99% of all requests is to be answered within 300 ms. Instability of performance can cause a delay of several tens of milliseconds. This can lead to SLO violations and reduced user experience, thereby having a negative impact on revenue. [12]. Tail latency is expressed in terms of percentiles; for instance, long tail latency implies latency as high as 99^{th} percentile.

It is difficult to keep the tail latency distribution short from the perspective of a service provider. This is due to the increase in the size or complexity of the interactive system and in total usage that subsequently affects tail latency. In addition, workloads that temporarily incur high latency often have a significant impact on the performance of the entire system.

Changes in response time result in high tail latency. The main reasons for the change in response time are as follows [8, 12, 13]:

- Shared resource: The server system shares key resources (such as CPU core, cache, memory bandwidth, and network bandwidth) with various applications. Even within the same application, depending on the operation, shared resources are used by different requests.

- Background daemons: Background daemons use only a small amount of resources on average, but increase the response time by several tens of milliseconds as tasks are allocated.

- Global resource sharing: Applications running on each machine use competitive global resources, such as network switches or shared storage.

- Power limits: Modern CPUs throttle the operating speed depending on their temperature. Therefore, if the CPU is active for a long time, it can affect the response time.

- Garbage collection: SSDs provide significantly fast random access, but require periodic garbage collection operations. This operation can increase read latency by more than 100 times.

- Energy management: Significant energy savings can be achieved using power saving mode on many devices. However, additional

latency occurs when switching from inactive to active mode.

- Timeouts: Failure tolerance and retry are widely used techniques in distributed systems. However, a single retry is sufficient to increase latency for the current request.

- Overload: The user sends large or too many requests. These requests can continue to use shared resources, such as CPU, memory, and network, which can queue up other requests.

- Maintenance activities: Background activities such as data reconstruction in a distributed file system, periodic log compaction, and periodic garbage collection in garbage-collected languages are the cause of periodic latency spikes.

The basic idea for reducing tail latency is hedging. Even when the task is paralleled, the slowest instance can be seen when the request is complete. For example, you can send more requests than you need and get the fastest return. In general, smaller task partitions also help to reduce tail latency.

Another approach to reduce tail latency is addressing the problem of head-of-line blocking. A few expensive requests increase the latency of cheap requests occurring simultaneously. Therefore, partitioning expensive requests into uniformly smaller tasks can help to reduce latency [8, 12, 13].

As mentioned earlier, there are several causes for tail latency. In this dissertation, we focus on reducing the long tail latency caused by garbage collection, which is an essential operation of SSDs.

2.2 Solid State Drive

2.2.1 Flash Storage Architecture and Garbage Collection

Figure 2.1 shows the internal architecture of an SSD. The flash memory chip package stores data and the DRAM is used for mapping table storage, data buffering and caching. The host interface (such as SATA and NVMe) exchanges data with the host using a pre-determined protocol. The flash memory controller schedules the flash memory accesses to maximize the parallelism of flash memory interface channels to boost the performance. The main processor controls the address translation and overall operation of the SSD.

Flash memory has the limitations of write-once (erase-before-write) and block erase. A flash translation layer (FTL) is used to overcome them, and various FTL algorithms have been studied. In recent years, page-level mapping [3] has been widely adopted to reduce write latency caused by the aforementioned flash memory characteristics, erase-before-write.

When writing new data in page-level mapping, the FTL allocates a new free page and then writes the data to the newly allocated page. The mapping information is updated between the logical and physical addresses. If the free block is insufficient (typically, if the number of free

blocks is less than a certain threshold), the FTL can obtain a new block through reclaiming, and this process is called garbage collection (GC). Figure 2.2 exemplifies the GC operation. As shown in the figure, the FTL reads the valid pages from the victim block and writes them to the newly allocated free block. This process is called valid page copy. When all the valid page copies from the victim block are completed, the victim block is erased. Through this process, a new free block is obtained. The page copies typically take much longer time than block erase. Thus, the GC latency is proportional to the number of valid pages of the victim block. In addition, the flash memory (to be exact, the target plane in flash memory) is blocked during the page copy operations, which increases the latency of subsequent requests thereby yielding the long tail latency problem. Generally, to minimize the valid page copy overhead, the victim block is selected as the block with the smallest number of valid pages. Recently, 3D flash memory, e.g., V-NAND has gained popularity. The 3D flash memory has more pages in a block than the 2D flash memory, which increases the latency of valid page copy and, finally, renders the problem of GC induced long tail latency more serious.

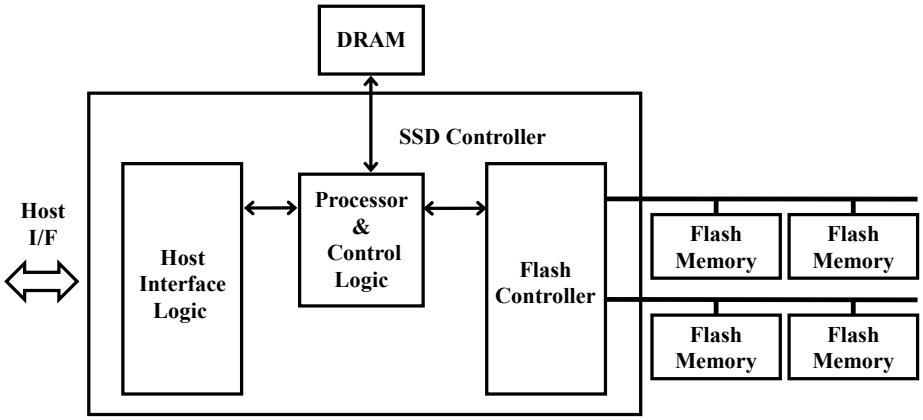


Figure 2.1 SSD internal architecture.

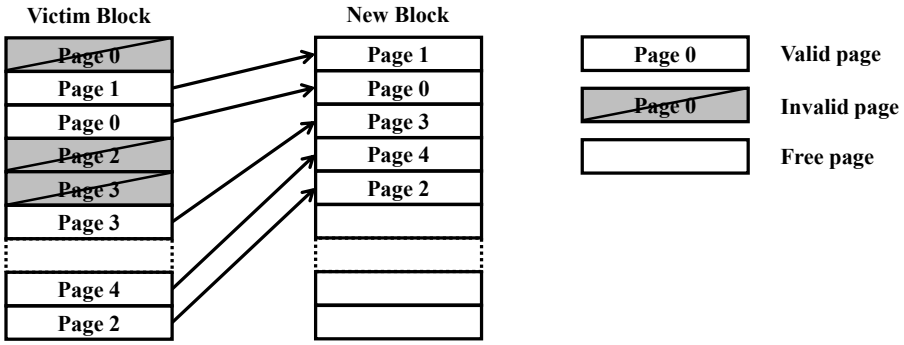


Figure 2.2 Garbage collection.

2.3 Reinforcement Learning

Figure 2.3 shows the environment-agent interaction of the RL model. The environment (storage system in this study) has states. The agent (GC scheduler in this study) selects an action (number of pages to copy) that maximizes all the future rewards expected from the current state of the environment. The environment executes the action received from the agent, passes the immediate reward (a function of latency) to the agent as a result, and switches to the next state. The RL model operates by repeating this process.

The basic components of the RL model are as follows:

State (S): The state can cover all the information of the environment (SSD system in this study) which is useful for the purpose of RL agent, i.e., reward maximization.

Action (A): The actions are the output of agent execution, which corresponds to the number of page copies to be performed in this study.

Reward (r): The reward is associated with the action. In this study, the shorter latency, the larger reward is given.

Policy (π): The strategy of the agent for selecting an action. Actions are selected to maximize the cumulative reward thereby minimizing long tail latency in this study.

Q-learning [14] is used in this study as a policy learning method. Q-learning updates the Q-value (all the future, i.e., cumulative reward)

using the value function of the state-action pair under the optimal policy, called Bellman equation, as follows:

$$Q(s,a) = E\{r_t + \gamma Q(s_{t+1}, a_{t+1}) | s_t = s, a_t = a\} \tag{2.1}$$

where s_t, a_t, and r_t represent the state and action at time step t and reward for a_t, respectively, and γ is the discount factor (set to 0.95 in our experiments). $Q(s,a)$ is the Q-value, i.e., cumulative reward expected when action a is taken at time t and state s.

The policy is defined as follows:

$$\pi(s) = argmax_a Q(s,a) \tag{2.2}$$

As shown in (2.2), the agent determines the action that maximizes the Q-value in state s. Q-learning is an on-line method, therefore, the policy is constantly modified, i.e., trained during runtime according to the dynamic behavior of the system.

We apply time-difference (TD) learning as follows.

$$Q(s,a) = Q(s,a) + \alpha\{r + \gamma Q(s',a') - Q(s,a)\} \tag{2.3}$$

where α is the learning rate (set to 0.3 in our experiments), r is the reward of the action a taken in state s, γ is the discount factor, and s' and a' are the next state and action, respectively [14]. The goal of the equation is to reduce the gap between the target Q-value, $r + \gamma Q(s',a')$ and current Q-value, $Q(s,a)$.

In (2.3), the Q-value on the Q-table, i.e., $Q(s', a')$ is re-used for a quick calculation of the Q-value update, which is called bootstrapping, wherein only the reward r is needed to update the Q-value. The equation eventually updates the policy because the policy is determined by the Q-value.

Q-learning uses a Q-table structure to store $Q(s, a)$ values. In order to build a Q-table, as examplified in Table 5.2 we need to bin the information of continuous values, e.g.,inter-request interval, into multiple levels. The size of the Q-table is the product of the number of state bins (in short, the number of states) and that of actions. In order to apply Q-learning to an embedded system such as the SSD, it is important to reduce the Q-table size.

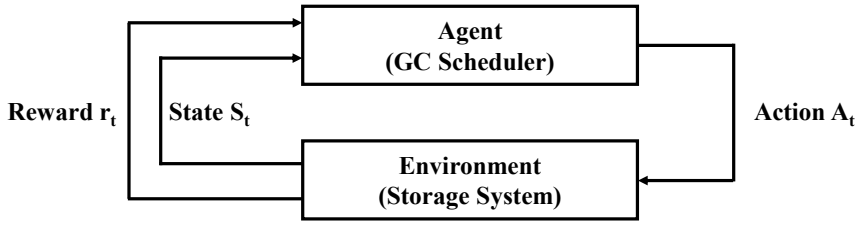


Figure 2.3 Environment-agent interaction.

Chapter 3

Related Work

Several studies have been conducted with the aim of reducing the latency induced by GC. Zhang et al. proposed a real-time lazy GC mechanism that adopts an on-demand page address mapping and a partial GC technique to improve the performance of the system [15]. In particular, partial GC divides the GC into several operations to ensure the worst system response time.

Chang et al. proposed a free-page replenishment mechanism wherein the real-time tasks were prevented from being blocked due to insufficient number of free pages. Assuming the write behavior of a realtime task is known, the number of GC operations and the maximum quantum for GC operation are determined to meet the real-time constraints [16].

Choudhuri et al. proposed GFTL, which helps perform partial GC to ensure fixed upper bounds in the latency of storage access by eliminating the source of non-determinism [17].

Qin et al. proposed a distributed partial GC policy in the RFTL, which tries to hide the long-tail latency due to the GC. Periodically, the method

helps perform partial GC and exploit buffer blocks to store the write data obtained during the GC operation, thereby reducing the GC-induced blockage [18].

Shahidi and Kandemir proposed cache-assisted GC technique (CachedGC) which resumes host requests immediately after moving valid page data to the DRAM buffer of SSD controller. Writing back the valid pages of DRAM buffer is performed when SSD has low utilization (i.e., idle time of SSD) [19]. However, they still have the problem of accurately estimating the idle time. In addition, a limited DRAM buffer is shared between the host write request and the GC. It means a reduction in the usable capacity of DRAM buffer for host write requests, which can affect write performance.

In [20], Wei et al. propose a workload-adaptive flash translation layer (WAFTL) with data partitions. It employs both page-level and block-level mapping blocks as normal data blocks. According to the data pattern, WAFTL selects the type of data block. The page-level mapping block handles random data and conducts partial data updates. The block-level mapping block stores sequential data. In particular, to reduce the garbage collection overhead, they utilize offline garbage collection to erase invalid blocks during idle time.

In [21], Yan et al. propose Tiny-Tail flash, which tries to eliminate tail latency due to garbage collection. They employ the four key techniques, plane-blocking GC, GC-tolerant reads, rotating GC and GC-tolerant flushes.

Plane-blocking GC reduces controller and channel blocking to plane blocking using a fine grained management scheme. GC-tolerant read prevents IO blocking due to the plane undergoing GC using a technique called RAIN, which exploits parity pages like redundant array of independent disks (RAID). Rotating GC helps to reduce IO blocking using a policy by which at most one plane in each plane group can run one GC at a time. GC-tolerant flush facilitates a rapid write buffer using capacitor-backed RAM.

A growing number of studies conducted with the aim of reducing storage-level tail latency. Amvrosiadis et al. present Duet, a framework that provides notifications about page-level events to maintenance tasks. The application uses these events as hints to process cached data. The tasks using Duet can finish maintenance work more efficiently due to they request fewer I/O operations. These opportunistic maintenance tasks require less I/O. Thus, tasks can complete faster. When tasks run concurrently, Duet helps to minimize affecting performance [22].

He et al. proposed Chopper, a tool to discover high-latency operation within local file systems. They focused on block allocation which is a critical contributor to uncommon behavior in recent systems. This could reduce file system performance. Chopper utilizes sophisticated statistical methodologies to discover the search space and diagnose intricate design problems efficiently. They pinpoint and remove four layout issues in ext4. Their improvements significantly reduce the problematic tasks causing

tail latencies [23].

Yang et al. proposed a split-level I/O scheduling framework that splits I/O scheduling logic across handlers at three layers of the storage stacks (block, system call, and page cache). Split schedulers can determine which processes issued I/O and accurately estimate I/O costs. This method can prevent file systems from striking orderings that are contrary to scheduling goals [24].

Studies have been conducted on approaches that utilize the idle time and workload prediction. Han et al. predicted the future workload and controlled the number of victim blocks [25]. The victim blocks are selected based on the age, utilization, and erase counts. The number of reclaimed blocks is then determined by predicting the history of the request count and rate.

Lin et al. predicted the future workload and obtained the number of victim blocks based on the predicted workload, erase count, and invalidation period [26].

Reinforcement learning has been widely used in a broad range of problems including robot control and resource allocation in data center. In [27], Ipek et al. proposed a self-optimizing DRAM controller design based on reinforcement learning. This memory controller sees the system state and predicts the long-term performance impact of each action it can perform. In this way, this controller learns to optimize its scheduling policy to offer maximum performance.

Pritzel et al. proposed a technique called neural episodic control, which can rapidly learn successful policies once they are underwent [28]. They employ a memory structure called differentiable neural dictionary (DND) that integrates slow-changing keys to quickly updating values. Thus, it utilizes context-based lookup on the keys. The DND is similar to Q-table cache [10] in that DND stores key-value pair and Q-table cache stores state-value pair. On the other hands, they manage memory in different ways. In the event of lookup, the DND gives a weighted sum of multiple values from the memory, whereas Q-table cache reads a single state-value pair. After the DND is read, a new key-value pair is saved into the DND. With regard to Q-table cache, in case of hit, Q-table cache updates the hit entry according to Q-learning and replacement policy. In case of miss, Q-table cache inserts a new entry based on replacement policy.

Mnih et al. proposed deep reinforcement learning (also called Deep Q-Network, in short, DQN) where a neural network is trained to produce Q-value output for the given input image [29]. Our proposed QP Net was inspired by this approach. Our key difference is that, instead of predicting Q-value with a single large network which is prohibitively expensive in embedded systems like storage, our scheme exploits the locality of storage behavior and, thus, consists of a small resource-efficient Q-table cache for short-term behavior and a small Q-value prediction network for long-term behavior. The DQN approach also employed an experi-

ence reply method to mitigate non-stationary distribution problem. The experience replay requires a large replay memory which can incur a large memory cost and may not be suitable in embedded systems with limited memory resource.

Konda and Tsitsiklis proposed an actor-critic method [30]. The actor network approximates the policy and outputs the probability of an action. The critic network approximates the target value function, namely the approximate optimal Q-value function. Our study was also motivated by the actor-critic method. Our solution is more resource-efficient in that we predict Q-value, i.e., the probability of action with a very small Q-table cache instead of a large actor network. Our QP Net is also small since it has only to predict the initial Q-value and a more accurate function of Q-learning is further performed on the Q-table cache.

Chapter 4

Small Q-table based Solution to Reduce Long Tail Latency

4.1 Problem and Motivation

4.1.1 Long Tail Problem in Flash Storage Access Latency

Figure 4.1 shows the latency comparison for a storage trace called *home2* (used in our experiments) between an ideal storage without a GC overhead and a real one with page-level mapping. The figure shows that the response time is short for the majority of the storage accesses. It is less than 1 ms for approximately 85% of the accesses. However, the latency difference between the median and the 99^{th} percentile is a factor of 100. As mentioned before, such a long-tail latency is a serious problem in real-time and quality-critical systems. For instance, the server storage typically needs to provide a minimum 7.5 ms of write latency for 99.99%

of the storage accesses. Considering that the GC latency continues to increase due to the increasing block size, it is important to reduce the long-tail latency for such real-time and quality-critical systems.

4.1.2 Idle Time in Flash Storage

Figure 4.2 shows the distribution of the request interval time for 60K requests the real-world workloads used in our experiments. The x-axis represents the inter-request interval time, and the y-axis represents the frequency of the request in each bin. As the figure shows, the storage system has frequent and long idle periods. Such an idle time can be exploited to perform GC operations. In idle time-aware GC methods [25, 26], it is important to determine how many GC operations need to be performed for a given idle time. The difficulty of this problem is that the length of the current idle period is unknown. To address this problem, several techniques exist [25, 26]. These techniques use fixed policies determined at the design time. Thus, they are limited in adapting to the dynamically changing storage access behavior because of the different program runs or phases.

In this dissertation, we propose an RL-assisted adaptive GC method [9], which learns the storage access behavior online and adjusts the GC to it to reduce the long-tail latency.

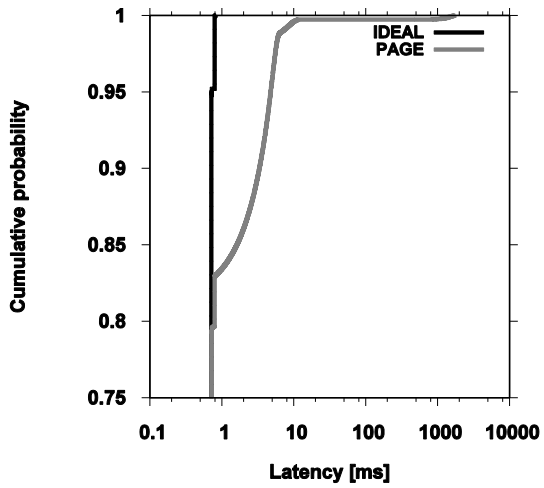


Figure 4.1 Long tail latency problem: *home2*.

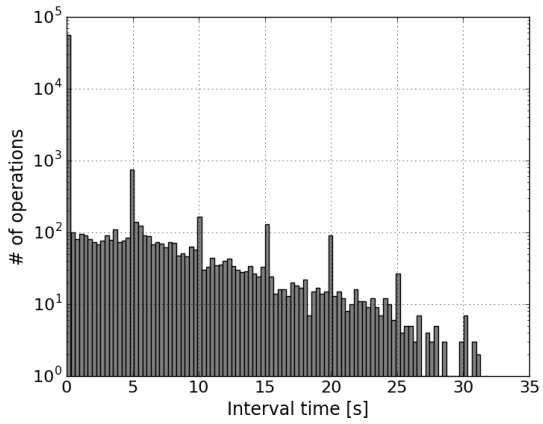


Figure 4.2 Inter-request interval distribution.

4.2 Design and Implementation

4.2.1 Solution Overview

We aim to reduce the long-tail latency by (1) hiding the GC latency by exploiting the idle time, and (2) minimizing the GC-induced blocking. In this section, we present an RL-assisted GC scheduler to hide the GC latency (Section 4.2.2) and an aggressive fine-grained partial GC scheme to reduce the blocking time (Section 4.2.3).

Our proposed RL-assisted GC scheduler is triggered in a lazy manner. Thus, only when an access request arrives at the storage and the number of free pages goes below a threshold [15], it is triggered. When triggered, it chooses an action. Because our GC method is based on the partial GC, the action is to perform a number of partial GC operations, e.g., five page copies from a victim block to a free block. Thus, the GC scheduler chooses an action, i.e., determines how many partial GC operations will be performed after serving the current request. An erase operation is performed when an action is chosen by the scheduler and a block is ready to be erased. In such a case, instead of executing the action, the block is erased.

After serving the request, the GC scheduler calculates the response time. Because our goal is to reduce the long-tail latency, we need to reflect the response time in our reward. We explain the details of how the

reward is calculated using the response time in Section 4.2.2. Note that the response time of the kth request gives the reward for the k-1th request. Thus, in the aforementioned Q-learning (Equation (3)), we update the Q value for the current state s and action a only after the next request is served and the corresponding reward is calculated.

4.2.2 RL-assisted Garbage Collection Scheduling

States: In the reinforcement learning, the states need to represent the history, which helps in maximizing the reward. We propose using the following information as the states.

- Previous inter-request interval
- Current inter-request interval
- Previous action

The inter-request interval is an important information of history because it reflects the intensity (i.e., the idleness) of storage traffics. Thus, if the interval is large, the RL-assisted GC scheduler tends to take a more aggressive action, i.e., more number of partial GC operations. The previous action plays a role of a summary of both recent history and the decision of the GC scheduler. From the viewpoint of the agent, both the host and the SSD subsystem constitute the environment. The inter-request intervals represent the state of the host. Note that the previous action can

represent that of the SSD subsystem as well as that of the host. It is because the previous action does not only plays a role of a summary of both recent history and the decision of GC scheduler, but also affects the state of the SSD subsystem, i.e., being busy in page copy or idle. For instance, if the previous action is to copy a large number of pages, then the current state of SSD subsystem tends to be busy.

We divide each of the three components into multiple bins, 2 bins for previous inter-request interval, 17 bins for current inter-request interval, and 2 bins for previous action, which gives a total 68 (=$2 \times 17 \times 2$) states. The details of binning are given in Section 4.3.1.

Reward: Regarding the reward, we need to assign a larger reward for a smaller response time. We also need to penalize an action giving a long response time. Figure 4.3 shows our reward function. The reward ranges between 0.5 and 1. For instance, if the response time is large (larger than the threshold t3), a negative reward is assigned to penalize the action. The thresholds in the reward function in Figure 4.3 need to be adjusted to the characteristics of the storage accesses. A fixed set of thresholds will not cover diverse scenarios in the storage accesses. Thus, we set the thresholds based on the characteristics of the storage accesses. In particular, we set three thresholds, t1, t2, and t3 to the 70^{th}, 90^{th}, and 99^{th} percentiles of the response time, respectively. Hence, even if the storage-access behavior changes, the thresholds can be adjusted based on the new distribution of the response time.

Exploitation and Exploration Balance: The exploration aims at filling in all the entries of the Q-table, and subsequently, improving them toward the optimal policy. To do that, we employ the ε-greedy technique [14]. In the initial period of RL execution (the first 1000 GC operations in our experiments), we utilize a large ε value (80%) to perform aggressive explorations. Then, we utilize a small ε value (1%) for a balance between exploitation and exploration during the rest of period.

GC Scheduling: Algorithm 1 shows the pseudo code of the proposed RL-assisted GC scheduler. For each request to the storage, the GC scheduler compares the number of free blocks N_{free} with threshold T_{GC} (=10 blocks in our experiments). If $T_{GC} >= N_{free}$, we call function e_greedy() (line 2), which performs either exploration or exploitation based on the probability of ε, i.e., a random action is selected at a probability of ε or an action is selected using the policy at a probability of 1 - ε [14]. Note that we do not trigger the GC scheduler in case of consecutive requests wherein the inter-request interval is zero (line 3–5). After serving the request and obtaining the response time for the current request (line 6), we perform the selected action, i.e., partial GC operation (line 7).We then call the reward function with the response time of the current request (line 8). Finally, we update the Q-table entry of the previous request (line 9). Note that, as mentioned previously, we update the entry of the Q-table associated with the previous request.

Intensive Garbage Collection: The baseline method in Algorithm 1

is not free from a blocked situation wherein the flash storage is out of the free block. To avoid such a situation, we employ an intensive garbage collection (GC) method from LazyRTGC [15] and modify it for further improvement. The objective of the intensive GC is to perform more (5 or 7 valid page copies in our experiments) partial GC operations than that in the normal partial GC operations (typically, 1 or 2 page copies), thus enabling faster reclamation of free blocks. The number (5 or 7) of partial GC operations is determined by considering the number of pages in a block and other parameters of the flash memory, e.g., erase time [15]. In [15], the intensive GC is triggered when there is only one free block left. Under the intensive GC, the action chosen by the RL policy is ignored and a fixed number of partial GC operations is performed after serving a write request. In [15], after the number of free blocks becomes greater than one, the intensive GC is no longer applied. In our work, we propose to utilize a larger threshold (termed the threshold of the stopping intensive GC, T^{I}_{GC}) than the one required to stop applying the intensive GC. We use a larger one (3), which is obtained via a sensitivity analysis in our experiments.

Algorithm 1: RL-Assisted GC Scheduling

Input: request, state$_{t-1}$(S_{t-1}), state$_t$(S_t), action$_{t-1}$(A_{t-1})

Output: action$_t$(A_t)

1 **if** $T_{GC} = N_{free}$ **then**

2 A_t = e_greedy(interval$_{t-1}$, interval$_t$, action$_{t-1}$)

3 **if** *interval$_t$ == 0* **then**

4 go to line 1

5 **end**

6 serve the request and obtain response time

7 run partial_gc(A_t)

8 r = reward(response_time)

9 $Q(S_{t-1},A_{t-1}) = (1-\alpha)Q(S_{t-1},A_{t-1}) + \alpha[r + (S_t,A_t)]$

10 **end**

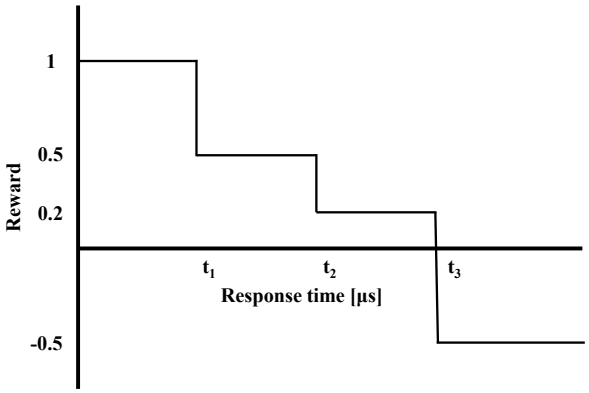


Figure 4.3 Reward function.

4.2.3 Aggressive RL-assisted Garbage Collection Scheduling

In this subsection, we propose two methods of aggressively triggering the GC to further reduce the long-tail latency [9]. To reduce the long-tail latency, it is effective to limit the maximum number of partial GC operations per action. In our experiments,we found that when the number of partial GC operations is limited to two, the best result is obtained. Thus, when the policy chooses an action, and if the action has more than two partial GC operations, we set the number of GC operations to two. When applying this method, we need to consider the blocking situation where the flash storage is out of the free block) because we limit the maximum number of partial GC operations.

To avoid the blocking situation, we trigger the GC collection more aggressively by introducing a new threshold for number of free blocks T^{A}_{GC}. T^{A}_{GC} is set higher than $T_{GC}(10)$. We call this method early GC triggering with the maximum limit of partial GC operation, in short, max-limited early GC triggering. Note that, the maximum number of partial GC operations is limited only when the number of free blocks N_{free} is between T^{A}_{GC} and T_{GC}. When $N_{free} <= T_{GC}$, the maximum limit is not applied to the action chosen by the RL-assisted GC scheduler.

The aggressive GC operation can increase the erase count. To avoid this, we carefully select the victim blocks. When N_{free} is within the two

thresholds T^{A}_{GC} and T_{GC}, we select a victim block only when it has a larger number of invalid pages than the threshold (60% of the block size in our experiments).

In conventional GC methods, a write request triggers GC when the number of free blocks is less than a certain threshold. In case of the read request, the GC is not triggered to avoid the increase in the read latency. We propose triggering a partial GC operation even for a read request when the triggering condition is met. Note that the latency of the read request does not increase because the GC operation is performed after serving the read request. We call this method read-initiated GC triggering.

Note that, in our aggressive method, the RL-assisted GC scheduler is triggered using the two methods: max-limited early GC triggering and read-initiated GC triggering. Based on our experiments, they prove useful in obtaining free blocks during the idle time, thereby reducing the long tail latency.

4.3 Evaluation

4.3.1 Evaluation Setup

We compare our proposed RL-assisted GC method [9] (baseline in Section 4.2.2 and aggressive in Section 4.2.3) with a typical GC method based on page-level mapping (page-level) [3] and LazyRTGC [15]. We implemented our proposed methods, page-level and LazyRTGC on a FlashSim simulator [31]. We use the metrics of long-tail latency at the 99^{th}, 99.99^{th}, and 99.9999^{th} percentiles and erase count. We use eight real-world workloads (six workloads from FIU [32] and two workloads from Microsoft [32]) and a synthetic one (from filebench [33]) as listed in Table 4.1. The goal of our work is to reduce long tail latency. In read-intensive workloads, the problem of long tail latency is not severe since GC is rarely invoked. Thus, we used write-intensive workloads in our experiments.

We started simulations with empty contents in the flash-memory model and measured the latency of all the requests for each workload.We use two types of 3D flash-memory systems as listed in Table 4.2.

Table 4.3 shows the binning for the components of the state. The binning was obtained by a sensitivity analysis on binning choices by varying the numbers of bins, 1∼3 and 15∼20 for previous and current inter-request intervals, and 1∼3 for previous actions, respectively, with an aim

to reduce the Q-table size, i.e., the number of states while improving the long tail latency. Considering that the accesses to NAND flash memory take 10∼1000µs, e.g., 49µs for read and 600µs for write [1], even though the agent is triggered at every storage access, the runtime overhead of the agent is negligibly small. It is because the agent accesses the Q-table (in a small SRAM) at maximum twice and executes a few instructions on the controller chip. Thus, the runtime of the agent is much smaller than the read latency of NAND flash memory.

Table 4.4 summarizes the thresholds used in our method. We obtained them by conducting a sensitivity analysis with all the storage traces. To improve the generality of our proposed methods, in our future work, we will investigate the feasibility of reducing the number of thresholds by enhancing the RL model, e.g., by introducing the number of free blocks into the states of the agent.

Table 4.1 Workload characteristics.

	Write ratio	Avg. interval [µs]	Avg. request size [KB]
home1	99%	85565	8.08
home2	91%	320548	9.40
home3	99%	1882329	8.26
home4	94%	693651	7.56
webmail	74%	303762	8.00
webmail+online	78%	127184	8.00
RBESQL	82%	11664	57.85
MSNSFS	67%	739	21.67
oltp	99%	84	4.46

Table 4.2 NAND Flash memories.

	3D 128 Gb	3D 512 Gb
Page size	8KB	16KB
Number of pages / block	384	768
Number of blocks / plane	2731	2874
Number of planes	2	2
Page read time	49 µs	60 µs
Page program time	600 µs	700 µs
Block erase time	4000 µs	3500 µs
Data transfer rate	533 Mbps	1 Gbps

Table 4.3 States.

Previous inter-request interval [μs]	Previous action	Current inter-request interval [μs]
< 100	< max action/2	< 100
		< 500
		...
		> 100000
	> max action/2	...
> 100	...	...
		...
	> max action/2	> 100000

Table 4.4 Thresholds.

Threshold	Value	Remark
T_{GC}	10	Triggering GC
T_{IGC}	3	Stopping intensive GC
T^{A}_{GC}	100	Triggering aggressive GC

4.3.2 Results and Discussion

Figure 4.4 compares the long-tail latency (in CDF) for writes. The figure shows that our proposed methods exhibit better long-tail latency than that using page-level and LazyRTGC. Page-level is not shown in the figure due to too large latency since it does not adopt any optimization to reduce long tail latency. LazyRTGC lies partial GC operations in a lazy manner and shows better latency than page-level.

Latency: Table 4.5 compares the latency normalized to LazyRTGC on a 3D 512Gb flash memory. Our baseline method (Base in the table) gives better (smaller) average latency: 0.86× at 99.9999^{th}, 0.94× at 99.99^{th}, and 0.92× at 99^{th} percentile. The gain is a result of the reinforcement learning-assisted action selection. LazyRTGC utilizes a fixed number of partial GC operations. In contrast, our proposed RL-assisted method can adapt to the characteristics of storage behavior, thereby providing variable number of partial GC operations to better exploit the idle time, which contributes to reducing the long-tail latency. Our aggressive method (Aggr in the table) gives much smaller latency: 0.76× at 99.9999^{th}, 0.71× at 99.99^{th}, and 0.92× at 99^{th} percentile. This proves that the two aggressive solutions, max-limited early GC triggering and read-initiated GC triggering, are effective in further reducing the long-tail latency.

In particular, the aggressive method gives much better latency in the

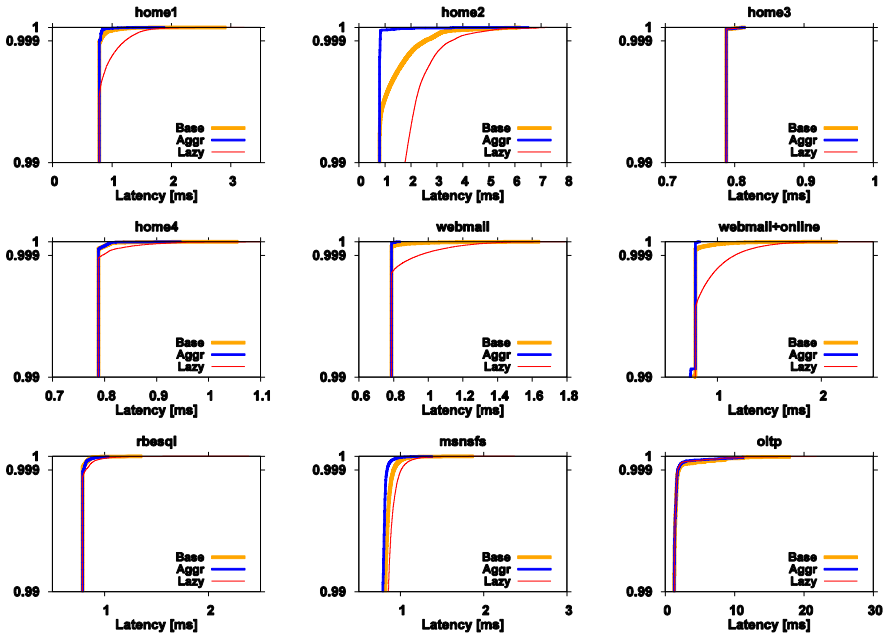


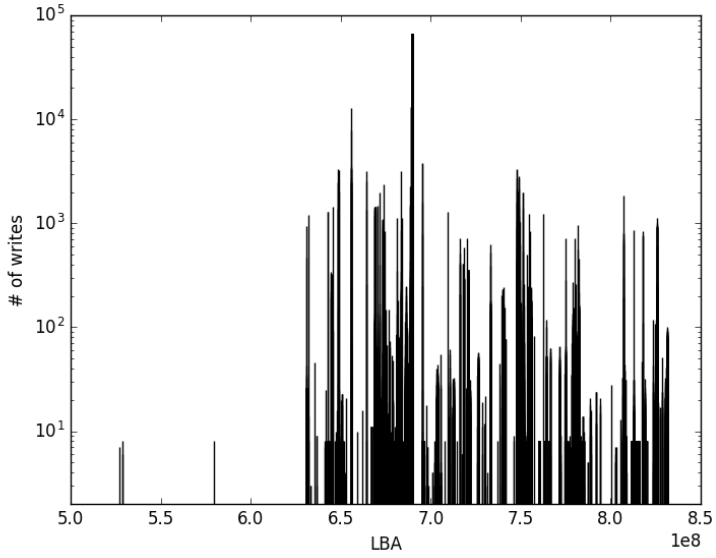
Figure 4.4 Comparison of write long-tail latency (3D 512Gb flash memory).

four workloads: *home1, home2, webmail, and webmail+online*. These workloads have heavy overwrite traffics distributed across a wide range of addresses. Figure 4.5 exemplifies the distribution of the write traffics for *home1* and *home3*. As the figure shows, in the case of *home1*, the overwrites are much stronger than that in *home3* (see y-axis). In addition, such strong overwrites are more distributed across a wider address range than that in *home3*.

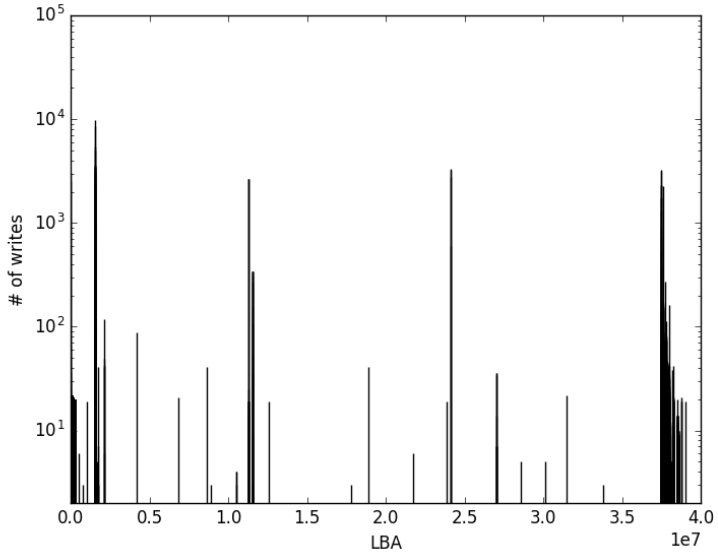
Such a write behavior in *home1* increases the ratio of invalid pages across a large number of blocks, which makes the GC cheaper, i.e., a free block can be obtained for fewer valid page copies. Thus, our aggressive method is effective in home1. However, as shown in Figure 4.5(b), *home3* has weaker overwrite behavior than *home1*, which makes it difficult for the aggressive method to reclaim the free blocks using fine-grained partial GC.

In Table 4.5, both the LazyRTGC and our methods give similar latencies in *home3* and *oltp*. In case of *home3*, the inter-request interval is large as listed in Table 4.1. In such a case, the GC (and its optimization) does not help in reducing the latency. On the other hand, *oltp* has very short idle time, i.e., small inter-request interval as listed in Table 4.1. Thus, there is little opportunity to improve the GC.

Table 4.6 compares the latencies in the case of a 3D 128Gb flash memory. Compared to the results in Table 4.5, our proposed methods give further reductions, e.g., 0.66× (in Table 4.6) vs 0.76× (Table 4.5),



(a) home1



(b) home3

Figure 4.5 Distribution of write traffics.

Table 4.5 Latency comparison on 3D 512Gb flash memory.

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVG
99.9999th	Page	465	239	N/A	962	514	697	9353	2246	33.1	1813
	Lazy	1.00	1.00	N/A	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	0.75	0.86	N/A	0.87	0.82	0.89	0.82	0.81	1.10	0.86
	Aggr	0.58	0.90	N/A	0.83	0.35	0.48	0.88	0.92	1.14	0.76
99.99th	Page	769	292	127	1105	679	848	6396	3435	62.9	1823
	Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	0.65	0.90	1.00	0.88	0.66	0.69	0.86	0.84	1.00	0.94
	Aggr	0.50	0.27	1.00	0.88	0.47	0.58	0.84	0.85	1.06	0.71
99th	Page	6.67	3.95	1.00	5.93	6.06	5.79	5077	10.5	2.91	568
	Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	1.00	0.44	1.00	1.00	0.98	1.00	0.95	1.00	0.99	0.92
	Aggr	1.00	0.44	1.00	1.00	0.93	1.00	0.94	1.00	0.98	0.92

Table 4.6 Latency comparison on 3D 128Gb flash memory.

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVG
99.9999th	Page	127	99	N/A	190	121	137	1007	717	1677	509
	Lazy	1.00	1.00	N/A	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	0.82	0.99	N/A	0.88	0.66	0.81	0.77	0.69	1.28	0.86
	Aggr	0.64	0.54	N/A	0.59	0.26	0.29	0.74	0.80	1.51	0.66
99.99th	Page	181	109	16.3	237	185	198	748	454	16.7	238
	Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	0.62	0.93	1.00	0.80	0.77	0.74	0.69	0.37	1.21	0.79
	Aggr	0.62	0.36	1.00	0.65	0.39	0.40	0.55	0.48	1.29	0.64
99th	Page	3.07	1.68	1.00	3.73	2.20	2.94	354	371	2.19	82.4
	Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	0.74	0.54	1.00	1.00	0.56	0.76	0.39	0.33	1.90	0.80
	Aggr	0.74	0.37	1.00	1.00	0.53	0.76	0.38	0.33	0.92	0.67

compared to the aggressive method at the 99.9999^{th} percentile. This is largely because of the low capacity of the 128Gb flash memory. The low capacity triggers GC more frequently, which increases the overhead of the GC in the conventional GC method (page-level). In Table 4.6, our proposed methods are more effective than the LazyRTGC in reducing the GC overhead in such a difficult condition.

Free block: Figure 4.6 shows the variation in the number of free blocks over time in the workload *home1* under LazyRTGC, and under our baseline and aggressive methods. As shown in the figure, after an initial period, LazyRTGC continues to retain 3 or 4 free blocks, which can lead to frequent GC operations because the number of free blocks is less. Our baseline method manages slightly more number (3–6) of free blocks. Our aggressive method manages significantly more number of free blocks, which helps in reducing the GC operations, thereby contributing to reducing the long-tail latency. Note that, as mentioned in Section 4.2.3, our aggressive method increases the number of free blocks only when there are victim blocks having a large ratio of invalid pages. Thus, although the aggressive method manages a significantly more number of free blocks than LazyRTGC, it does not have a negative impact on the erase count, as demonstrated later in this Section.

Erase Count: Tables 4.7 and 4.8 compare the erase counts (normalized to LazyRTGC) on 512 Gb and 128Gb flash-memory systems, respectively. From Tables 4.7 and 4.8, it is clear that our proposed aggres-

sive method and LazyRTGC give similar erase counts while the page-level gives a higher erase count because of the block-level GC.

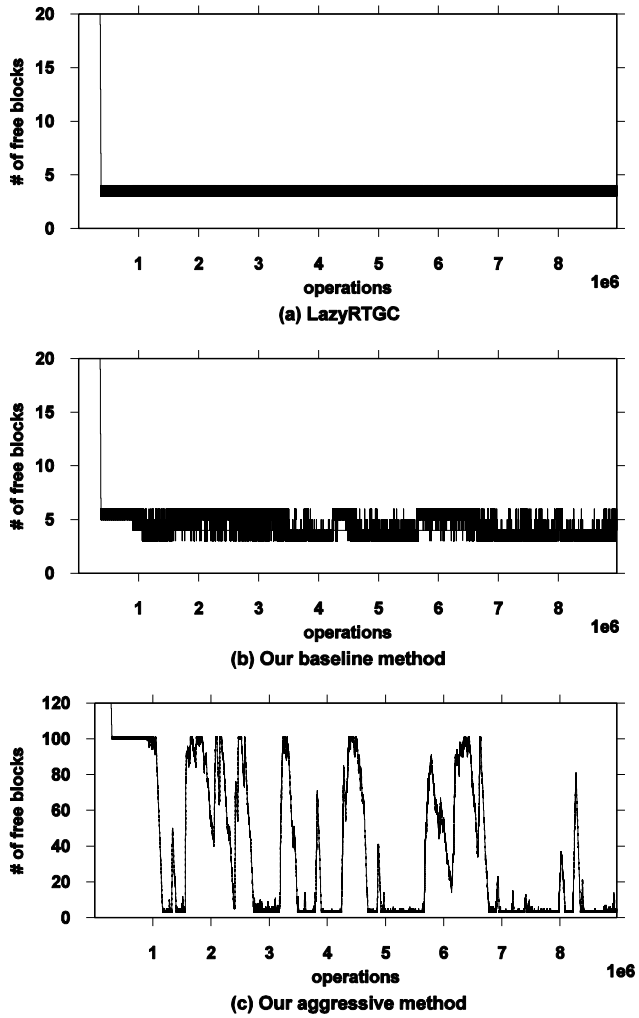


Figure 4.6 Comparison of number of free blocks.

Table 4.7 Erase count comparison on 3D 512Gb flash memory.

	home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVG
Page	1.92	1.33	1.50	1.83	1.59	1.69	10.9	2.07	1.67	2.72
Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Base	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Aggr	0.91	1.00	1.02	1.01	1.02	1.03	1.03	0.95	1.01	1.00

Table 4.8 Erase count comparison on 3D 128Gb flash memory.

	home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVG
Page	1.26	1.16	1.26	1.56	1.19	1.41	0.56	0.63	1.81	1.20
Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Base	0.92	0.97	1.00	1.01	0.98	1.01	0.93	0.97	3.15	1.21
Aggr	0.93	1.00	1.06	1.12	1.01	1.1	0.94	0.98	1.14	1.03

RL related Analysis: In order to evaluate the robustness of our method, we measured the latency of ten executions of each trace. Tables 4.9 and 4.10 show that the results of proposed method are consistent having a very small standard deviation of latency, 3.8% of the average normalized latency.

We evaluated the utilization of Q-table entries for each workload. In the analysis, we found that the average utilization is 79% and there is possibility of further improvement by adjusting the Q-table size to each workload, which is left for our future work.

Average application performance: It is important to evaluate the impact of our proposed method on average application performance. Since we did trace-based experiments, the average latency of request is considered to be correlated with average application performance. Our experiments (the corresponding results of which are omitted due to page limit) show that, the average latency of our proposed baseline and aggressive methods is slightly better than that of the existing method, LazyRTGC. Thus, we can state that our proposed methods improve the long tail latency without degrading the average application performance.

Long trace experiment: We also evaluated our proposed method with longer traces by stitching the original traces. Our experiments show that, in the long trace cases, our proposed method outperforms the existing one, lazy as in the case of short ones.

Simple prediction method: Our problem of reducing long tail la-

tency could be addressed by existing, possibly simpler, alternatives such as those based on time series prediction. In our experiments, we did a quantitative comparison with GC methods based on two typical methods of predicting the inter-request interval with moving average and exponential smoothing, respectively. Tables 4.11 and 4.12 show that our proposed method constantly outperforms them. It is because our method manages history and learns appropriate actions in a more fine-grained manner using the Q-table.

In summary, the experimental results show that the LazyRTGC does not fully utilize the idle time available in the storage workload. In contrast, our baseline method can better exploit the idle time because of the reinforcement learning-based GC. In addition, our aggressive method helps in further reducing the long-tail latency by (1) preparing free blocks with frequent small fine-grained partial GCs, which helps in reducing the frequency of triggering the GC operations and stalling the subsequent requests, and (2) hiding the GC operation by exploiting the idle time based on the reinforcement learning. Consequently, as presented in Tables 4.5 and 4.6, our proposed aggressive method helps in reducing the long-tail latency by 29–36% at the 99.99^{th} percentile for the two flash-storage devices.

Table 4.9 Standard deviation of normalized latency on 3D 512Gb flash memory.

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVERAGE
99.9999th	Base	0.055	0.051	0.000	0.031	0.030	0.043	0.038	0.042	0.055	0.038
	Aggr	0.026	0.049	0.000	0.027	0.002	0.002	0.026	0.029	0.008	0.019
99.99th	Base	0.017	0.072	0.000	0.006	0.015	0.022	0.003	0.002	0.003	0.016
	Aggr	0.014	0.052	0.000	0.000	0.000	0.000	0.003	0.002	0.005	0.008
99th	Base	0.000	0.000	0.000	0.000	0.004	0.000	0.000	0.000	0.003	0.001
	Aggr	0.005	0.000	0.000	0.000	0.004	0.000	0.000	0.000	0.001	0.001

Table 4.10 Standard deviation of normalized latency on 3D 128Gb flash memory.

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVERAGE
99.9999th	Base	0.047	0.047	0.000	0.036	0.033	0.053	0.027	0.046	0.044	0.037
	Aggr	0.025	0.111	0.000	0.048	0.018	0.000	0.025	0.029	0.000	0.028
99.99th	Base	0.011	0.035	0.000	0.027	0.006	0.013	0.005	0.004	0.065	0.018
	Aggr	0.006	0.070	0.000	0.016	0.000	0.000	0.004	0.005	0.000	0.011
99th	Base	0.000	0.009	0.000	0.000	0.000	0.000	0.000	0.000	0.009	0.002
	Aggr	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000

Table 4.11 Latency comparison of simple prediction method on 3D 512Gb flash memory.

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVERAGE
99.9999th	Aggr	0.58	0.90	N/A	0.83	0.35	0.48	0.88	0.92	1.14	0.76
	Mov10	0.97	1.01	N/A	1.01	0.67	0.60	0.97	0.70	1.56	0.94
	Mov100	1.11	1.02	N/A	1.01	0.90	0.95	0.99	0.76	1.36	1.01
	Exp0.1	0.87	1.02	N/A	0.93	0.84	0.90	0.97	0.63	1.56	0.97
	Exp0.3	0.97	1.01	N/A	1.00	0.65	0.69	0.99	0.59	1.62	0.94
99.99th	Aggr	0.50	0.27	1.00	0.88	0.47	0.58	0.84	0.85	1.06	0.71
	Mov10	0.61	0.98	1.00	0.95	0.58	0.71	0.98	0.84	1.12	0.86
	Mov100	1.00	1.05	1.00	0.95	0.97	0.99	0.97	0.82	1.12	0.99
	Exp0.1	0.84	1.04	1.00	0.94	0.84	0.92	0.99	0.82	1.12	0.95
	Exp0.3	0.66	0.99	1.00	0.95	0.64	0.78	0.98	0.82	1.08	0.88
99th	Aggr	1.00	0.44	1.00	1.00	0.93	1.00	0.94	1.00	0.98	0.92
	Mov10	1.00	0.75	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.97
	Mov100	1.00	0.75	1.00	1.00	1.00	1.00	0.99	1.00	0.99	0.97
	Exp0.1	1.00	0.71	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.97
	Exp0.3	1.00	0.72	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.97

Table 4.12 Latency comparison of simple prediction method on 3D 128Gb flash memory.

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVERAGE
99.9999th	Aggr	0.64	0.54	N/A	0.59	0.26	0.29	0.74	0.80	1.51	0.66
	Mov10	0.93	0.99	N/A	0.94	0.76	0.84	0.79	0.64	1.50	0.92
	Mov100	1.53	3.51	N/A	1.41	1.05	1.09	0.72	0.68	1.41	1.43
	Exp0.1	1.16	1.08	N/A	1.21	1.01	1.07	0.72	0.61	1.59	1.06
	Exp0.3	0.96	0.99	N/A	1.04	0.74	1.00	0.84	0.54	1.66	0.97
99.99th	Aggr	0.62	0.36	1.00	0.65	0.39	0.40	0.55	0.48	1.29	0.64
	Mov10	0.77	0.95	1.00	0.98	0.81	0.85	0.86	0.63	1.15	0.89
	Mov100	0.99	1.68	1.00	1.03	1.08	1.08	0.85	0.57	1.32	1.07
	Exp0.1	0.94	1.01	1.00	1.03	1.07	1.06	0.86	0.56	1.13	0.96
	Exp0.3	0.85	0.96	1.00	0.98	0.98	1.02	0.86	0.56	1.13	0.93
99th	Aggr	0.74	0.37	1.00	1.00	0.53	0.76	0.38	0.33	0.92	0.67
	Mov10	0.87	0.83	1.00	1.00	0.84	0.94	0.93	1.00	0.92	0.93
	Mov100	1.07	0.86	1.00	1.00	1.09	1.11	0.93	1.00	0.92	1.00
	Exp0.1	1.00	0.86	1.00	1.00	0.97	0.99	0.93	1.00	0.93	0.96
	Exp0.3	0.89	0.84	1.00	1.00	0.83	0.89	0.93	1.00	0.92	0.92

Chapter 5

Q-table Cache to Exploit a Large Number of States at Small Cost

5.1 Motivation

Techniques which apply RL have been studied in an effort to reduce long-tail latency induced by GC [9]. In the RL model, multiple states are used to represent the environment (e.g., the workload characteristics and SSD internal information). Using appropriate states which represent the given environments, which we call key states, is essential to obtain successful RL assisted solutions.

Based on the work [9], we conducted experiments and found that the long-tail latency varies according to the number of states. Figure 5.1 shows the 99.9999^{th} percentile latencies when the number of states is increased in *home1* which is one of workloads we used in experiments (as described in Section 4.2.3). The results show that increasing the number of states (by taking finer bins in this case) tends to decrease the latency. On the other hand, as shown in the figure, increasing the number

of states does not always improve the performance consistently, as the design space of most RL-assisted applications is not always monotone or linear. This is closely related to the information used as states and how they can be divided into multiple bins.

In addition, we investigated locality behavior of the RL solution. To do this, we selected four periods in the middle of the workload, *home2*, at equal periods (10,000 requests per period). We registered which states were used and counted how many times they were accessed during the period. Table 5.1 shows the top ten states sorted by access count and the access counts at four periods in *home2*. The table shows that the states used for each period are different. The access counts are also different for each period. For example, the top two states account for the majority of access counts in period 2. However, other states have relatively low access counts, indicating that the characteristics corresponding to the two top ranked states account for a large proportion in this period.

On the other hand, the access counts of the upper ranked states are not high during period 4, indicating a greater variety of characteristics than in the other periods. We continued the experiment with various workloads, observing that different workloads have different states access patterns, though this data is not shown here due to page limit.

Through the above two observations, we realized that using more states is essential to ensure better performance and that creating a generalized solution, which adapts to the dynamic behavior, is required.

In [9], RLGC uses Q-learning [14], which is a type of RL method. In Q-learning, reward values (Q-value) for all stateaction pairs are stored in a table called Q-table. That is, as the number of states increases, the size of Q-table also increases. In the SSD firmware, there is a stringent constraint on the code size. Thus, the desired solution needs to be dynamic to adapt to the changing behavior of SSD system, and allow for a large number of states while reducing the cost of Q-table, i.e., keeping a small Q-table.

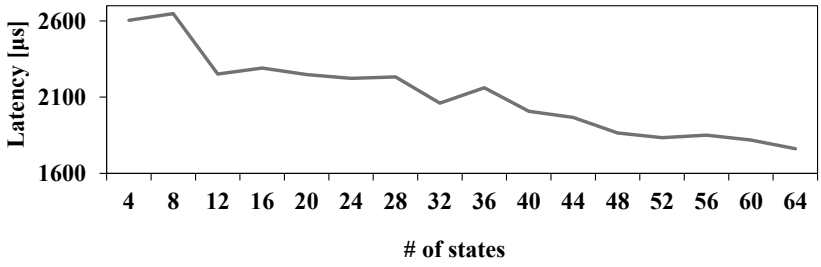


Figure 5.1 Latency variation according to the number of states in *home1*.

Table 5.1 Top rank states and access counts in *home2*.

Period 1		Period 2		Period 3		Period 4	
State #	Count	State #	Count	State #	Count	State #	Count
199813153	3152	199803970	5779	424000545	1246	274455586	123
349109313	963	349133890	2524	199822369	328	200025122	88
274455585	853	424000545	132	423757951	321	199803969	88
423760929	849	199804036	55	423831585	279	199914530	86
199887871	627	423907361	49	274621473	181	199803938	85
199969825	593	423969825	48	423757857	127	199803937	79
199803937	543	199804063	35	199960609	122	274454562	77
199886881	464	199804003	24	274454563	111	199969857	77
274482209	323	199804899	22	199831585	98	274474049	73
349189153	300	199803999	22	274455585	84	274537506	73

5.2 Design and Implementation

5.2.1 Solution Overview

The purpose of this study is to reduce long-tail latency through the (1) dynamic management of states appropriate for the characteristics of the environment, including the properties of the workload and the SSD internal status, and (2) to reduce I/O blocking.

The proposed technique [10] is based on RLGC [9]. RLGC uses partial GC which is used in LazyRTGC [15], as a GC method. To exploit the inter-request interval (idle time), RLGC employs RL. Figure 5.2 shows the agent and environment interaction in the RL solution. The agent (the GC scheduler in our study) is a decision-maker that determines an action. At time t, the agent receives state st and reward r_t (response time, e.g., the write latency of the previous request in our study) from the environment (the storage system in our study). The agent selects action at (the number of partial GC instances to be performed in our study) according to the learned policy and sends it to the environment. The environment takes an action from the agent and then passes the next state s_{t+1} and reward r_{t+1} to the agent. The agent learns and updates the policy using the reward received from the environment.

We use Q-learning [14] as a policy learning method of RL. Q-learning manages the value function (representing the expectation of cumulative

reward when taking the action at the state) for the state-action pair and updates the value function using the state-action pair and reward. The value function pertaining to the state-action pair is as follows,

$$Q(s,a) = E\{r_t | s_t = s, a_t = a\} \tag{5.1}$$

where s(s_t) and a(a_t) are the state and action at time t, respectively, and r_t is the reward at time t. Q-value $Q(s,a)$ is the expectation of cumulative reward when the environment takes action a at time t. After an action is taken and the associated reward is available, the Q-value is updated as a weighted sum of current Q-value and the most recent reward for the state-action pair [9].

The policy to determine the action is as follows:

$$\Pi(s) = argmax_a Q(s,a) \tag{5.2}$$

In equation 5.2, the policy determines an action which maximizes the Q-value at state s. In terms of RL implementation, the important data structure is the Q-table, which stores the Q-values. The number of Q-table entries is states $\times$ actions. When adopting RL-assisted solution in embedded systems, small Q-tables are required. The RL agent learns behavior of the workload, and determines GC triggering time point and how many partial GC instances to be executed as an action. RLGC [9] uses a fixed set of states obtained from three types of predefined infor-

mation binned into 68 states at design time. Due to the fixed small set of states, RLGC has significant limitations in achieving further reduction in long tail latency, which requires a large number of states and an adaptation to the dynamic behavior of the workload to be applied to storage as explained in our observations.

The proposed method [10] uses a dynamic key states management technique to overcome the limitations of RLGC. We use approximately 88×108 state candidates from binning and a combination of 17 pieces of information. As a means of storing the Q-value for the state-action pair, we use a small Q-table cache that uses an eviction policy in the least recently used (LRU) manner instead of the Q-table of the previous RLGC. We use a Q-table cache per action. Our method has also three actions depending on the number (0, 1 and 2) of page copies in partial GC, as in the aggressive (aggr) method of RLGC. Thus, three Q-table caches are used for the three actions to be executed. As the storage system works, the state is determined and the reward is calculated according to the taken action. The state and corresponding Q-value pair are stored in the Q-table cache. If the determined state is already in Q-table cache, the Q-value is updated. On the other hand, if the determined state does not exist in the Q-table cache, the Q-value is added as a new entry. At this point, if there is no more free entry space in the Q-table cache, one of the existing entries is evicted according to the LRU policy. Through this process, the proposed RL-assisted GC scheduler further reduces the

long-tail latency while using a small Q-table cache, i.e., small memory resource. In Section 5.2.2, we explain the dynamic key state management method in detail.

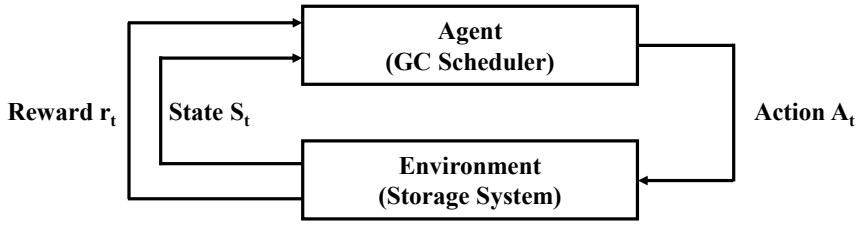


Figure 5.2 Environment and agent interaction.

5.2.2 Dynamic Key States Management

As mentioned earlier, the proposed technique is based on the aggressive method of RLGC [9]. Therefore, we focused on the difference to the RLGC.

Candidate States: In order to present the characteristics of various workloads, we need a considerable number of candidate states. Table 5.2 shows the information used in relation to the states and the number of bins for each piece of information. The combination of 17 pieces of information and their binned states yield 88 $\times$ 108 states. This helps to reflect more fine-grained status changes as compared to the use of only 68 states in the previous RLGC approach. Note that we selected the information listed in the table through extensive analyses of candidate information. However, more information can be used when applied to an actual SSD product.

Storing the Q-value: As noted earlier, a very large amount of memory space is necessary when using a Q-table to store a large number of Q-values for state-action pairs. For example, to use a Q-table for 88 $\times$ 108 states listed in Table 5.2, we need a prohibitively large table of 98GB (=88 $\times$ 108 (states) $\times$ 3 (actions) $\times$ 4B). It is impractical to apply this into an embedded system such as SSD firmware.

In order to solve this problem, we use a Q-table cache that stores only an active subset of the state and Q-value pairs. Figure 5.3 shows the

organization of the Q-table cache used in our experiments. In this study, we use three Q-table caches each with 100 state-Q value pair entries. The memory space for the three Q-table caches is only 2.34KB (= 100 (entries) $\times$ 2 (state, Q value) $\times$ 3 (actions) $\times$ 4B) taking a negligible amount of memory space in a SSD. This Q-table cache stores a certain number of state-Q value pairs. As mentioned before, if there is no free space into which to add new entry, an existing entry is evicted according to the LRU policy.

When receiving a new request, the GC scheduler (agent) runs as follows. First, each of the three Q-table caches is looked up with the current state as the typical data cache in computer architecture is looked up with an address. Then, if there is any match, then the action having the maximum Q value is selected and executed according to Equation 5.2. The Q value is calculated using the reward, i.e., the latency of the previous request. In case of miss, the state-Q value pair is inserted in the Q-table cache for the action selected at time $t-1$. If the state-Q value pair already exists in the Q-table cache for the action at time $t-1$, the Q value is updated and marked as the most recently used entry. More details about how an action is determined are given in the following subsection.

Eviction policy: The purpose of using the Q-table cache to store the state-Q value pair is to store key states representing the environment properly using a small amount of memory space. Meaningful states can reflect recent behavior of the workload. This helps cope with workload

behavior changes quickly. It can also reflect frequently and repeatedly occurring behaviors. To deal with these two factors, LRU and a least frequently used (LFU) policy were considered. However, the LFU is impractical when attempting to manage the history of the frequency of accesses for all stored entries [34]. Thus, in this study, we use the LRU policy only as an eviction policy. Although our solution uses only the LRU policy, we gain a further reduction of the long-tail latency (as described in Section 5.3.1).

Figure 5.4 shows a histogram of the number of states corresponding to the state access frequency range in *home1*. The x-axis represents the range of access count, and the y-axis the number of states. For an intuitive understanding of this, the x-axis uses the three different interval scales of 1 for 1 to 10, 10 for 11 to 100 and 1000 for 101 to 10100. Figure 5.4 shows that there is a clear distinction between states with very low and high access frequencies. Among all states which occur in *home1*, the number of states that occur only once account for a significant portion. On the other hand, Figure 5.5 shows the state statistics obtained after running the workload, i.e., at the end of workload. This exemplifies that, at an instant of workload run, the Q-table cache can have a high percentage of frequently accessed states, i.e., a large number of meaningful states. Note that because the process by which state-Q value pair is evicted and newly added is repeated under the LRU eviction policy, some states with an access frequency of 1 can exist.

Action determination: Our GC scheduler selects an action having the maximum Q value. The difference relative to Q-learning using a complete Q-table arises when the state-Q value pair used to determine an action cannot be found in the Q-table cache. This occurs when the state-Q value pair is not yet added to the Q-table cache or the state-Q value pair was already evicted. If the GC scheduler cannot find the Q value for a given state in all Q-table caches, it then selects action 0 (no GC). This is a conservative approach to lower the possibility of increased latency due to a lack of information.

Table 5.2 State information and # of bins.

Information used for state	# of bins
Current (t) inter-request interval	32
Previous (t-1) inter-request interval	32
Previous (t-1) action (# of performed partial gc)	3
Previous (t-2) action (# of performed partial gc)	3
Previous (t-3) action (# of performed partial gc)	3
Previous (t-4) action (# of performed partial gc)	3
Previous (t-5) action (# of performed partial gc)	3
# of free blocks	12
Previous (t-1) request size	5
Previous (t-2) request size	5
Previous (t-1) valid page copy (performed or not)	2
Previous (t-2) valid page copy (performed or not)	2
Previous (t-1) block erase (performed or not)	2
Previous (t-2) block erase (performed or not)	2
Current (t) requested operation	2
Previous (t-1) requested operation	2
Previous (t-2) requested operation	2

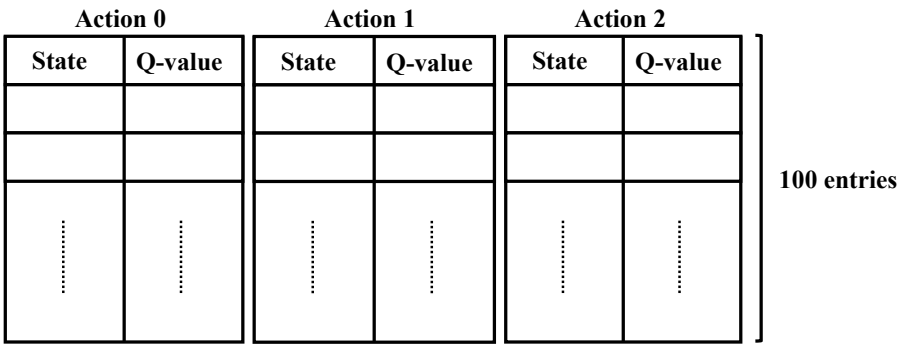


Figure 5.3 Q-table cache architecture.

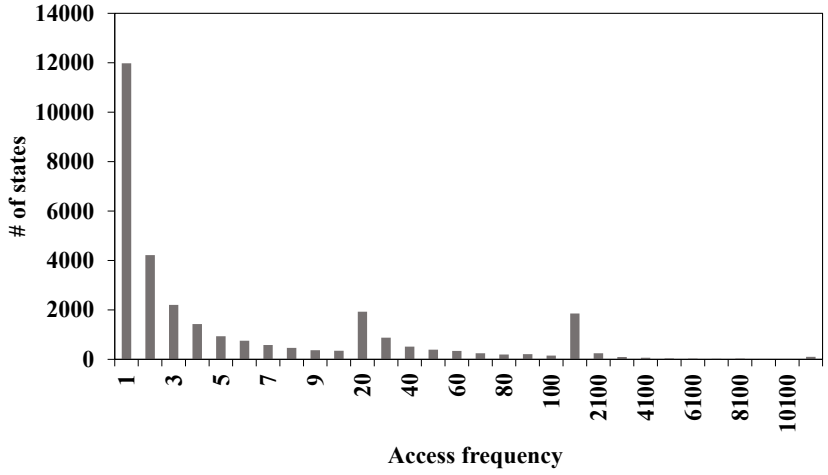


Figure 5.4 Number of states for each access frequency.

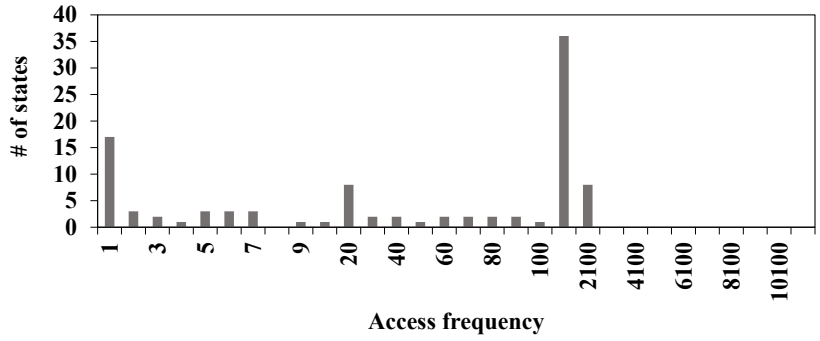


Figure 5.5 Number of states for each access frequency (after running the workload).

5.3 Evaluation

5.3.1 Evaluation Setup

We compared the proposed dynamic key states management technique [10] with an aggressive method which shows the best performance in RLGC [9]. We implemented the proposed method and RLGC [9] using the popular FlashSim simulator [31]. We use the 99^{th}, 99.99^{th} and 99.9999^{th} percentiles of latencies as the metrics. Ten real-world workloads (*home1, home2, home3, home4, webmail+online, webmail, MSNSFS, RBESQL, TPCC,* and *TPCE*) [32] and one synthetic workload [33] are used in our evaluation. Two types of 3D flash memory are used. Table 4.2 shows the detailed parameters of the two types [1, 2]. Each Q-table cache has 100 entries to store the state-Q value pairs. To generalize our solution, we present the results when using the Q-table cache with 100 entries as a representative result. We also report on the effects of the Q-table cache size.

5.3.2 Results and Discussion

Latency: Table 5.3 (a) shows the latency comparison with RLGC on 512Gb of 3D flash memory. The results are normalized to RLGC. The proposed method has a good (low) average latency over the baseline (aggressive method of RLGC), with 0.78× at the 99.9999^{th}, 0.88× at the

99.99^{th}, and $0.98\times$ at the 99^{th} percentiles. The baseline learns the policy using a small number of states. On the other hand, our method utilizes a much larger number of fine-grained states and manages key states among them. As a result, it is possible to obtain better (lower) latency with information useful for selecting actions and with fine-grained GC scheduling.

There was almost no improvement in the latency for the three workloads of *home4, webmail+online* and *webmail*. These results are already close to the intrinsic program time of flash memory. Thus, there is no additional room for further reduction. For *home3* and *oltp*, the baseline and proposed method give similar results. *home3* has a long inter-request time, and long-tail latency due to GC accordingly does not occur. For *oltp*, it has a very short inter-request interval; thus, there is little opportunity for optimization [9].

Table 5.3 (b) shows the results on 128Gb of 3D flash memory. It shows better (lower) average latency, e.g., 25% reduction at the 99.9999^{th} percentile, than the results with 512Gb of 3D flash memory. It is because the capacity of 128Gb is less than that of 512Gb, more frequent GC is required. Therefore, there is more opportunity for further optimization.

Q-table cache entry size: Table 5.4 shows the latency variation for four workloads when the number of Q-table cache entries changes. The notation "100 of Ours_100" refers to the number of Q-table cache entries. The results of *home1* and *home2* show the best (lowest) latency in Ours_100. On the other hand, *MSNSFS* and *RBESQL* show the best (low-

est) latency in Ours_2000. As noted earlier, in order to generalize the proposed method, we applied a common Q-table cache size which shows the best average performance improvement for all workloads. Therefore, we used a Q-table cache size of 100 entries.

Table 5.5 compares the total number of states. In the table, the values of *MSNSFS* and *RBESQL* are greater by ten times than those of *home1* and *home2*. This indicates that the workloads create a large number of states because they undergo significant behavior changes. This explains why the two workloads work better with large Q-table caches in Table 5.4.

Table 5.6 shows the hit rate of Q-table cache for each of the workloads with a Q-table cache size of 100. As discussed earlier, *MSNSFS* and *RBESQL* have more significant behavior changes than the other workloads. Thus, the Q-table cache hit rates are lower relative to the other workloads. These results also demonstrate the possibility of optimizing the Q-table cache size, which is left as future work.

Erase count: Tables 5.7 and 5.8 compare the erase counts on 512Gb and 128Gb of 3D flash memory. These results are normalized to the baseline. In both cases, the erase counts are similar to the corresponding baselines.

Table 5.3 Latency comparison.

Percentile		(a) 3D 512Gb												(b) 3D 128Gb											
		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	TPCC	TPCE	AVERAGE	home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	TPCC	TPCE	AVERAGE
99.9999th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Ours	0.32	0.23	1.00	0.99	1.00	0.99	1.03	0.95	1.04	0.20	0.82	0.78	0.36	0.34	1.00	0.71	0.98	0.98	1.01	0.97	0.92	0.12	0.82	0.75
99.99th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Ours	0.82	0.61	1.00	1.00	0.99	0.98	1.02	1.04	1.04	0.18	0.99	0.88	0.53	0.63	1.00	0.89	1.00	1.00	0.97	1.03	0.88	0.19	0.98	0.83
99th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Ours	1.00	1.00	1.00	1.00	1.00	1.00	1.03	1.00	0.99	0.77	1.00	0.98	0.98	1.00	1.00	1.00	1.01	1.00	1.00	1.00	0.96	0.66	1.00	0.96

Table 5.4 Latency comparison for various Q-table cache size on 3D 512Gb flash memory.

		home1	home2	MSNSFS	RBESQL
99.9999^{th}	Base	1.00	1.00	1.00	1.00
	Ours_100	0.32	0.23	1.03	0.98
	Ours_200	0.32	0.25	1.05	0.93
	Ours_500	0.79	0.31	0.97	0.93
	Ours_1000	0.77	0.41	0.93	0.94
	Ours_2000	0.99	0.47	0.92	0.93
	Ours_3000	0.81	0.44	1.06	0.98

Table 5.5 Number of states visited for each workload on 3D 512Gb flash memory.

	home1	home2	MSNSFS	RBESQL
# of states	37039	35428	444782	211395

Table 5.6 Hit rate of Q-table cache in each workload on 3D 512Gb flash memory.

	home1	home2	MSNSFS	RBESQL
Hit rate	81.78%	75.94%	33.51%	45.82%

Table 5.7 Erase count for 3D 512Gb flash memory.

	home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	TPCC	TPCE	AVERAGE
Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Ours	1.13	1.01	1.00	1.00	1.08	1.05	1.03	0.99	1.00	0.99	1.00	1.03

Table 5.8 Erase count for 3D 128Gb flash memory.

	home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	TPCC	TPCE	AVERAGE
Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Ours	1.12	1.03	1.00	1.00	1.08	1.00	1.02	1.00	1.00	0.99	1.00	1.02

Chapter 6

Combining Q-table cache and Neural Network to Exploit both Long and Short-term History

6.1 Motivation and Problem

6.1.1 More State Information can Further Reduce Long Tail Latency

In the RL model [9], multiple states are used to represent the dynamic behavior of environment. Using the appropriate state to better represent the environment is essential for a successful RL based solution. We call such a state key state. Based on the work [9], we experimentally found that the long tail latency varies with the number of states. Figure 5.1 shows the latency of SSD at the 99.9999^{th} percentile when we increase the number of states in *home1*, one of the workloads we used in our experiments. As a result, increasing the number of states by taking finer bins (to obtain

a Q-table) tends to decrease the latency. However, increasing the number of states does not always reduce latency consistently. This is because the design space of most RL assisted applications is not always linear or monotonic. This is closely related to what information is used as a state and how to divide them into multiple bins to build the Q-table.

6.1.2 Locality Behavior of Workload

We analyzed the locality behavior of the RL based solution [9]. To do this, we chose four equal length periods (10,000 requests per period) in the middle of *home2*, one of the workloads we used in our experiments. We recorded what states were used during this period and counted the frequency of accessing each state. Table 6.1 shows the top twelve states sorted by access count. The table shows that the statistics of access counts is highly skewed while showing locality. For example, period 2 exhibits a strong locality behavior since top two states dominate access counts while other states have relatively small access counts. The table also shows that the states used in each period are different, which means the state access behavior changes over time. Our idea of Q-table cache was motivated by the observations that system behavior exhibits locality which changes over time.

6.1.3 Zero Initialization Problem

Using more state information has the advantage of expressing the environment in details, and can further reduce the long tail latency as shown in Figure 5.1. The Q-table cache can use many state candidates in a small memory space [10]. However, the evicted state from the Q-table cache loses the previously learned Q-values. Thus, it relies on the zero initialization of Q-values for newly inserted state-Q value pairs in the Q-table cache. The zero initialization increases the learning time of newly inserted states and finally degrades the quality of action choice on such states.

In order to mitigate the problem of zero initialization, the size of the Q-table cache may be increased thereby reducing the loss of the learned Q-values of the evicted states. However, the large Q-table cache can incur prohibitively high memory cost. For instance, in order to keep all the states (88×10^8 states) shown in Table 5.2, the Q-table size could reach 98GB, which is prohibitively expensive in embedded systems such as the SSD (described in Section 6.2.2).

Especially, a large Q-table cache tends to contain a large number of states that are not sufficiently learned, which we call *immature states*. As will be explained later, such immature states can yield inappropriate action choices, which prevents us from lowering latency.

Table 6.1 Top rank states and access counts in *home2*.

Period 1		Period 2		Period 3		Period 4	
State #	**Count**	**State #**	**Count**	**State #**	**Count**	**State #**	**Count**
199813153	3152	199803970	5779	424000545	1246	274455586	123
349109313	963	349133890	2524	199822369	328	200025122	88
274455585	853	424000545	132	423757951	321	199803969	88
423760929	849	199804036	55	423831585	279	199914530	86
199887871	627	423907361	49	274621473	181	199803938	85
199969825	593	423969825	48	423757857	127	199803937	79
199803937	543	199804063	35	199960609	122	274454562	77
199886881	464	199804003	24	274454563	111	199969857	77
274482209	323	199804899	22	199831585	98	274474049	73
349189153	300	199803999	22	274455585	84	274537506	73
274454561	76	274455523	19	274538529	77	423815202	71
200025121	58	199804900	17	423760929	77	423942239	65

6.2 Design and Implementation

6.2.1 Solution Overview

Figure 6.1 shows the overall architecture of the proposed solution which integrates the Q-table cache [1] and a small neural network called Q-value prediction network (QP Net) [11]. The Q-table cache learns the short-term behavior to select actions on the given state. On the contrary, the QP Net is trained to learn the long-term behavior of the system. Since the QP Net has the global view of Q-function, it can provide good initial Q-values in case of inserting new entries to the Q-table cache.

Such an integration of QP Net with the Q-table cache offers a low-cost high-performance implementation of RL-based solution to the SSD. As will be shown in the experiments, both Q-table cache and QP Net incur very small cost while learning the Q-function of key states and the global view of Q-function. Finally, the integrated solution offers further reduction in long tail latency than the cases that a small Q-table is used [9] and only the Q-table cache is utilized [10, 11].

The overall operation of our solution which shown in Figure 6.2 is as follows. First, the SSD receives a request from the host and starts serving it. While serving the request, e.g., writing data to the flash mem-

[1] In the original Q-table cache solution [10], three Q-table caches each with 100 state-Q value pair entries are used. Q-table cache for each action is operated independently to prevent loss of learned information, e.g., Q-values. In this dissertation, we propose Q-table cache where each entry consists of one state and three Q-values for three actions.

ory, the RL agent applies (2.2), i.e., updates the Q-table cache (updating the entry corresponding to the previous action or inserting a new state-Q value pair) and QP Net while utilizing the information of reward and new state. Then, the RL agent chooses an action based on the new state. After finishing the service of the current request, the RL agent executes the chosen action, e.g., 1-page copy. The above steps are repeated. Note that our proposed GC solution is triggered only when the SSD receives write requests.[2]

There are two aspects regarding idle time. First, our solution aims at exploiting idle time by executing the action during idle time. This is because the idle time can come after finishing the request. Second, however, our solution does not fully exploit idle time, especially, very long idle time since the partial GC operation is triggered only when a request arrives at the SSD. Exploiting long idle time will be a promising topic in our future work.

In the following subsections, we describe how the components of the RL agent, i.e., the Q-table cache and QP Net are constructed and performed in detail.

[2]In the original Q-table cache solution [10], the partial GC, selected by the RL agent, is performed at each SSD access (read and write). In this dissertation, we propose applying the GC solution only after the service of write request. It is mainly because our new integrated solution aims at hiding the latency overhead of QP Net execution by the write latency. Our experiments will compare the two cases of GC execution: on read/write request in [10] and only write request in the proposed integrated solution.

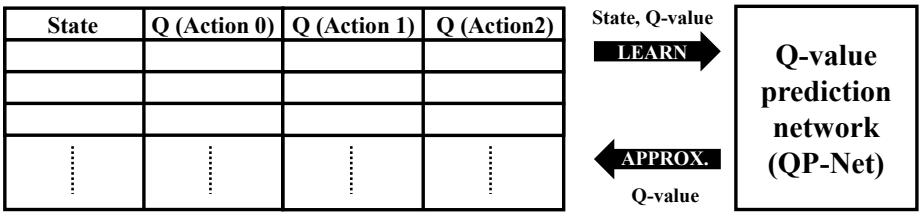


Figure 6.1 Q-table cache with QP Network.

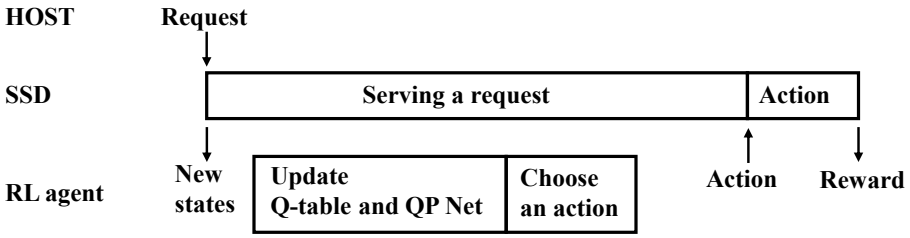


Figure 6.2 Operation overview.

6.2.2 Q-table Cache for Action Selection

The Q-table cache [10, 11] runs as a Q-table which determines actions on the given state and updates its Q-values on the given reward. Compared with RLGC [9], the Q-table cache tries to manage a much larger set of states with a small cache structure. In this subsection, we describe how the states are constructed, how the Q-values are accessed and updated in the cache structure, and how the cache replacement is performed.

• **States:** In order to represent the characteristics of various workloads in detail, a considerable amount of state information is needed. RLGC [9] uses a total of 68 states obtained by binning from three pieces of information defined at design time. Such a small amount of state information prevents RLGC from further reducing the long tail latency. Table 5.2 shows the information used for the states and the number of bins for each piece of information used in our work. Such a large amount of state information helps to reflect more detailed state changes. Note that we selected the information shown in Table 5.2 through extensive experiments. We think that, when our scheme is applied to actual SSD products, more product-specific information can be added to the state information.

• **Q-table Cache:** As mentioned earlier, storing a large number of state-Q value pairs in a Q-table requires a very large amount of memory. For example, in order to store the 88×10^8 states mentioned in Table 5.2, the Q-table will require approximately 98GB (= 88×10^8 (states) $\times$

3 actions $\times$ 4B) memory space. It is impossible to apply such a solution to an embedded system such as an SSD firmware.

In order to address this problem, we employ a Q-table cache that stores only the active subset of states. Figure 6.1 shows the structure of Q-table cache. In this study, as shown in the figure, we use the Q-table cache which consists of one state and three Q-values for three actions. Note that we utilize only three actions, 0, 1 or 2 page copy in the RL solution. We selected three actions through sensitivity analysis. The memory capacity of the Q-table cache is 1.56KB (= 100 (entries) $\times$ 4 (state and actions) $\times$ 4B). This is a negligible amount of memory cost in an SSD.

The RL agent operates on the Q-table cache as follows. First, upon receiving the current state, the agent looks up the Q-table cache to see if the current state is found in the three Q-table caches. To do this, we try to find a match by comparing the current state and those in the Q-table cache. If there is a match, then we select an action among 0/1/2-page copy, based on the policy to be explained below.

For each action, its associated reward is obtained by applying a reward function. Basically, we assign the larger reward when the smaller latency is obtained as the result of action. Figure 6.3 shows the reward function used in the proposed method.[3] To be exact, we measure the response time of the request immediately subsequent to the action. Then, we obtain the reward of the action utilizing the reward function in the

[3]We obtained the reward function by extensive experiments.

figure. As shown in the figure, a large reward is assigned to a small response time, which favors actions which incur small latency in subsequent requests. Likewise, by assigning a negative reward for a long response time, the actions incurring long latency are penalized. After obtaining the reward, the Q-value associated with the action is updated in the Q-table cache using (2.3).

In case of miss in the Q-table cache, we need to perform cache replacement. If there is no free entry in the cache, we select a victim entry under the LRU policy. Then, we insert into the Q-table cache a new entry corresponding to the current state. We will explain how to initialize the Q-values of the new entries in Section 6.2.3.

• **Handling Negative Reward:** Note that it is important to handle negative reward in case of action 0. Action 0, which performs zero page copy, does not run the GC. Even if action 0 is selected, the latency of the subsequent request may be increased due to workload behavior (i.e., heavy write requests with short inter-request time), which may result in a negative reward of action 0. In this case, the long latency is not the result of the selected action (action 0). In such a case, the previous work [10] assigned negative reward to the action, which will penalize the action. In this work, as Figure 6.3 shows, we propose assigning zero reward, instead of negative one, to the action, which avoids penalizing such an action [11].

• **Policy for Selecting an Action:** In order to select an action, we

adopt the deterministic policy of (2.2) under ε-greedy method [14]. Basically, we select an action that maximizes the Q-value in (2.2).[4] In addition to the deterministic policy, we also adopt ε-greedy method in order to balance between exploitation and exploration. Thus, we select an action in a random manner at the probability of ε while applying the deterministic policy of (2.2) at the probability of 1-ε.

Note that, compared with the conventional policy, our Q-table cache can have a unique situation when there is a miss in the Q-table cache. In such a case, the agent selects 0-page copy as a default action. This is a conservative approach to avoid the possibility of increased latency due to the deficiency of information.

6.2.3 Q-value Prediction

We propose a neural network for Q-value prediction [11] in order to solve the zero initialization problem of the original Q-table cache [10].

• **Q-value Prediction Network Architecture:** Figure 6.4 shows the architecture of Q-value prediction network (QP Net) [11]. The QP Net is a multi-layer perceptron (MLP). We chose the two-layer MLP as the QP Net architecture based on the fact that a two-layer MLP can approximate any arbitrary nonlinear function [35–37]. As mentioned earlier, our idea was motivated by the critic model in the actor-critic model. We also

[4]The selection of action having the maximum Q-value is deterministic compared with the statistical policy where the action is selected in a probabilistic manner in proportion to the associated Q-value.

considered that the critic model is typically designed with the MLP. In order to determine the specific configuration of the two-layer MLP, we performed sensitivity analysis where we varied the number of neurons on each hidden layer, which will be given in our experiments (Section 6.3.4). The input layer receives 17 inputs, which are the normalized values of the 17 pieces of information used in the candidate states described in Table 5.2. The output layer has three outputs, whose output values are the predicted Q-values for actions 0, 1, and 2-page copy.

We train the QP Net whenever the Q-table cache is updated. It is because the QP Net needs to learn the whole behavior of Q-function. The QP Net is similar to the critic network of actor-critic model [30] in that both are trained whenever a new reward is available. The QP Net trained for the whole behavior of Q-function is advantageous especially when unseen states occur and their initial Q-values need to be predicted as well as when previously evicted states are re-inserted to the Q-table cache. If the QP Net were trained only for the evicted entries of Q-table cache, it could not make useful predictions on unseen states.

Note that the QP Net is trained with all the three Q-values of the Q-table cache though only one of the three Q-values is updated. Such a training enables the QP Net to learn the relative importance between actions as well as state-dependent Q-values. When inserting a new entry to the Q-table cache, we utilize all the three Q-values predicted by the QP Net to initialize the three Q-values of the newly inserted state.

• **Pre-training QP Net:** We train the QP Net during runtime together with the Q-table cache. We observed that a randomly-initialized QP Net gives poor performance. It is mainly because the randomly-initialized QP Net can give, as the prediction, large random Q-values, especially in the beginning of the system run. Such large initial Q-values hurt Q-learning by making the Q-table cache training difficult. In order to resolve this problem, we experimented with pre-training the QP Net with random and real traces and found pre-training with random traces is effective because the QP Net pre-trained with random traces tends to give Q-value predictions with much smaller variations than that pre-trained with real traces. In our experiments, we pre-train the QP Net during design time with 10,000 random requests. We also experimented pre-training with real traces (as explained in Section 6.3.4). However, the pre-training with random traces outperforms that with real ones.

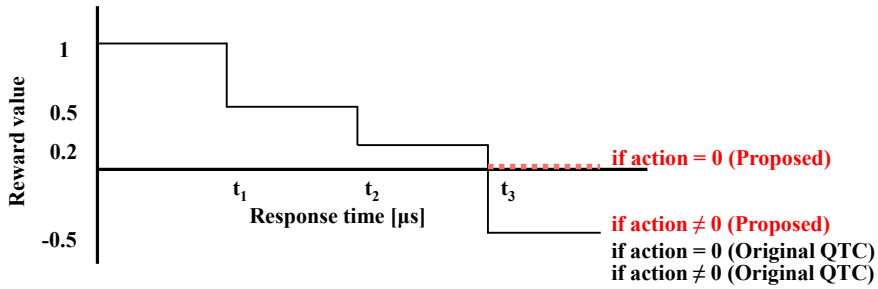


Figure 6.3 Reward function.

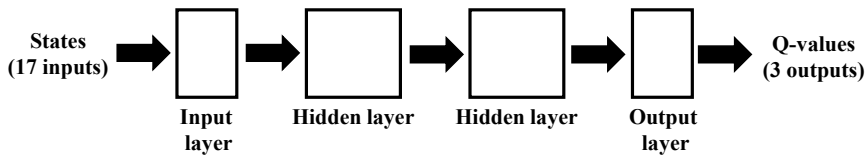


Figure 6.4 QP Net architecture.

6.3 Evaluation

6.3.1 Evaluation Setup

We compare the proposed method with RLGC (aggressive scheme) [9] which shows the best performance (lowest latency) among the techniques applying reinforcement learning to GC. We implemented the proposed method and RLGC using a widely-used flash storage simulator, Flashsim [31].

We used 16 Open Storage traces (*home1*, home2*, home3*, webmail+online*, webmail*, MSNSFS*, RBESQL*, RA*, DADS*, DAP*, DTR*, MSNSCFS*, homes*, online*, webresearch*, and webuser**) [38] which are re-played SNIA block traces [32] with NVMe SSD array and collected from storage nodes. We also used 13 real-world traces (*home1, home2, home3, home4, webmail+online, webmail, MSNSFS, RBESQL, TPCC, TPCE, EXCH24, Financial1 and Financial2*) [32] and a synthetic one (*oltp*) [33]. Experiments were conducted on flash memory types of 3D 512Gb [2] and 3D 128Gb [1]. The parameters of the two flash memory types are shown in Table 6.2. The latency at the 99.9999^{th}, 99.99^{th} and 99^{th} percentile was used as the evaluation metric of the experiment. The QP Net [11] was initialized as described in Section 6.2.3.

Table 6.2 Characteristics of flash memories for 3D 128Gb [1] and 3D 512Gb [2].

Parameters	**3D 128Gb**	**3D 512Gb**
Number of planes	2	2
Number of blocks / plane	2731	2874
Number of pages / block	384	768
Page size	8KB	16KB
Page read time	49 µs	60 µs
Page program time	600 µs	700 µs
Block erase time	4000 µs	3500 µs
Data transfer rate	533Mbps	1Gbps

6.3.2 Storage-Intensive Workloads

In the course of the experiment, it was observed that some workloads were already sufficiently optimized by Q-table cache, and there was no room to further reduce the long tail latency. Thus, we identified storage-intensive workloads, where we evaluate the proposed method.[5]

To do this, we compared the Q-table cache method with the NO GC case. The NO GC case assumes that flash memory is over-writable. This means that GC does not occur. Thus, the minimum latency can be obtained in this case [11].

Figures 6.5 and 6.6 compare Q-table cache method and NO GC case in the 3D 512Gb and 3D 128Gb flash memory types, respectively. The experimental results were normalized to NO GC cases and the results were sorted in descending order. We assumed that storage-intensive workloads should exhibit sufficiently (10% in our experiments) longer latency than the NO GC case. In the case of the 3D 512Gb flash memory, as shown in Figure 6.5, the latency of Q-table cache method is similar to that of the NO GC case (within 10%) in 17 workloads (*home2, home1, home4, webmail, webmail+online, TPCE, home3*, home3, home2*, DADS*, DAP*, MSNSCFS*, homes*, online*, webresearch*, webuser* and Financial2*). For the 3D 128Gb flash memory, as shown in Figure 6.6, the difference between Q-table cache method and NO GC cases is less than

[5]Identifying intensive workloads and evaluating on them is a popular approach, especially in memory sub-system architectures [39–42].

10% in 19 workloads (*webmail*, webmail+online*, home4, home1*, webmail+online, webmail, home1, TPCE, home3, home2*, home3*, DADS*, DAP*, MSNSCFS*, homes*, online*, webresearch*, webuser* and Financial2*). Therefore, we performed the following experiments only for the storage-intensive workloads (having 10% larger latency than NO GC case).

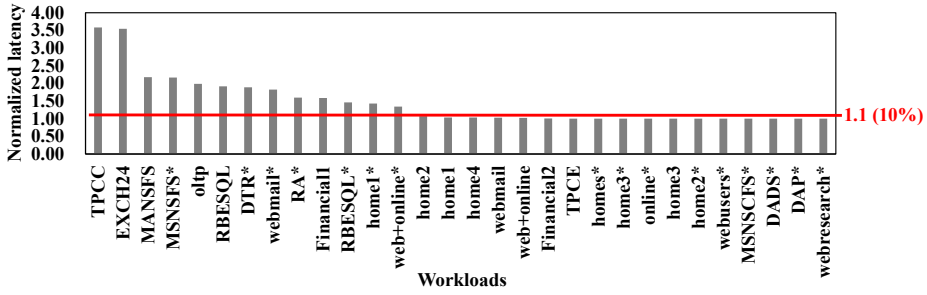


Figure 6.5 Latency comparison with NO GC case at 99.9999^{th} percentile on 3D 512Gb flash memory.

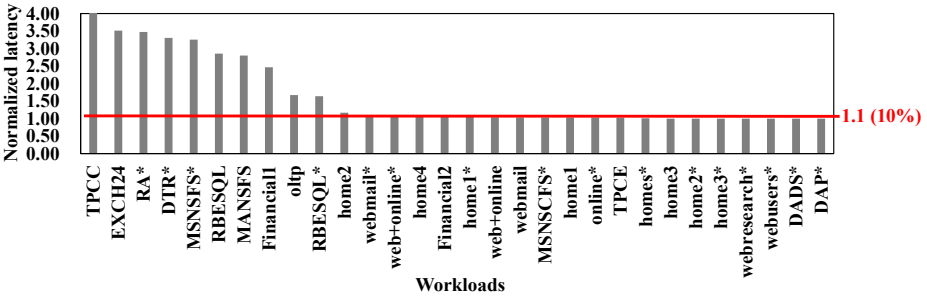


Figure 6.6 Latency comparison with NO GC case at 99.9999^{th} percentile on 3D 128Gb flash memory.

6.3.3 Latency Comparison: Overall

Table 6.3 shows a latency comparison with the RLGC (**Base** in the table) [9] on the 3D 512Gb flash memory. The results are normalized to the RLGC. Our proposed method has two approaches, Q-table cache only (**QTC** in the table) [10], and the integrated method of Q-table cache and QP Net (**QPN** and **QTCW** in the table)[6] [11].

Both of our proposed methods (QTC and QPN in the table) show better (lower) average latency than the baseline. QTC gives latency reductions by 0.89× at the 99.9999^{th} percentile, 0.90× at 99.99^{th} and 0.98× at 99^{th}. QPN offers further reductions by 0.75× at 99.9999^{th}, 0.80× at 99.99^{th} and 0.98× at 99^{th}.

The baseline uses a small number of states to learn policy. However, QTC exploits a much larger number of fine-grained states and maintains key states among them. Thus, it offers smaller latency than the baseline.

Our integrated solution of Q-table cache and QP Net (QPN in the table) gives further latency reductions by training QP Net during runtime to provide better initialization of Q-table cache than the zero initialization of the original Q-table cache, which finally contributes to better action selection. Note that the latency improvement of QPN comes from better Q-value initialization since both the QTC and QPN utilize the same

[6] As we mentioned earlier, we propose applying the GC solution only after the service of write request to integrated solution. To verify the effect of not applying GC on read request, we provided the results of Q-table cache only method at write request only (**QTCW** in the table). The results are explained later in this subsection.

number of candidate states.

Table 6.3 also compares the latency on the 3D 128Gb flash memory. Our methods show better (lower) average latency than the baseline by 0.78×(QTC)/0.63×(QPN) at the 99.9999^{th} percentile, 0.85×(QTC)/0.69×(QPN) at 99.99^{th}, and 0.97×(QTC)/0.92×(QPN) at 99^{th}. That is, the QP Net gives additional 14-15% reductions to the original Q-table cache [10] in the two types of flash memory.

The results of the 3D 128Gb flash memory show more latency reduction than those of the 3D 512Gb flash memory. It is because, the 3D 128Gb flash memory requires more frequent GCs because it has smaller capacity than 3D 512Gb flash memory, which is also confirmed in Figures 6.5 and 6.6 where the 3D 128Gb flash memory gives larger latency in the intensive workloads than in the 3D 512Gb flash memory.

In particular, in the 3D 128Gb flash memory, our integrated solution shows much larger latency reduction in the two workloads, *RA** and *MSNSFS** than in other workloads. It is because they have lower Q-table cache hit rates than others. As shown in Table 6.4, the hit rates of *RA** and *MSNSFS** are 0.31 and 0.48, respectively. Therefore, in the case of Q-table cache only (QTC), their Q-table caches suffer from frequent evictions and insertions with zero Q-value, which amplifies the negative impact of zero-initialized Q-value on action selection. QP Net in our integrated solution improves this situation with better initial Q-values thereby enabling better action choices, which finally leads to lower la-

tency.

Table 6.3 shows QPN gives the largest latency reduction in *TPCC*. It is because *TPCC* has a very large latency (above 90,000μs) as compared to other workloads. Hence, there is a large potential of further optimization as shown in Figures 6.5 and 6.6. Thus, our method could reduce the normalized latency down to 0.02$\times$ in the 3D 128Gb flash memory. However, the latency reduction becomes less in the 3D 512Gb flash memory, 0.15$\times$. This is because the latency of the 3D 128Gb flash memory is around 3$\times$ longer than that of the 3D 512Gb flash memory, which shows that the 3D 128Gb flash memory has more potential of further latency reduction.

In addition, Table 6.3 compares the latency of QTC (Q-table cache only method at each SSD access) and QTCW (Q-table cache only method at write request) as well. As shown in the table, both QTC and QTCW give similar average latency reductions which are 0.89$\times$(QTC)/0.91$\times$ (QTCW) at 99.9999^{th} percentile, 0.90$\times$(QTC)/0.91 $\times$(QTCW) at 99.99^{th} and 0.98$\times$ (QTC)/0.99$\times$(QTCW) at 99^{th} in the 3D 512Gb flash memory. In case of the 3D 128Gb flash memory, QTC also gives similar average latency to QTCW. It is because, both of QTC and QTCW method use a large number of state candidates to represent the environment. Note that our integrated solution (QPN) adopts QTCW to overlap the QP Net latency with the flash memory write operation.

Table 6.3 Latency comparison.

Percentile		(a) 3D 512Gb (50 neurons per hidden layer)														(b) 3D 128Gb (100 neurons per hidden layer)											
		TPCC	EXCH24	MSNSFS	MSNSFS*	oltp	RBESQL	DTR*	webmail*	RA*	Financial1	RBESQL*	home1*	web+online*	AVERAGE	TPCC	EXCH24	RA*	DTR*	MSNSFS*	RBESQL	MSNSFS	Financial1	oltp	RBESQL*	home2	AVERAGE
99.9999th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.20	0.98	1.03	1.02	1.04	0.95	0.93	0.93	0.70	1.00	0.92	1.00	0.88	0.89	0.12	0.87	0.68	0.86	1.01	0.97	1.01	0.96	0.92	0.85	0.34	0.78
	QTCW	0.25	0.98	1.03	1.02	1.04	0.95	0.95	0.96	0.73	1.00	0.93	1.00	0.94	0.91	0.18	0.90	0.69	0.88	1.01	0.98	1.01	0.96	0.92	0.87	0.35	0.80
	QPN	0.15	0.82	0.93	0.87	1.00	0.88	0.81	0.61	0.47	0.78	0.86	0.78	0.75	0.75	0.02	0.80	0.31	0.79	0.62	0.84	0.88	0.65	0.88	0.80	0.33	0.63
99.99th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.18	1.01	1.02	1.01	1.04	1.04	0.93	0.87	0.99	1.00	0.85	0.93	0.79	0.90	0.19	0.94	0.96	0.89	0.98	1.03	0.97	0.97	0.88	0.87	0.63	0.85
	QTCW	0.20	1.01	1.03	1.01	1.04	1.05	0.93	0.90	1.00	1.00	0.85	0.94	0.86	0.91	0.23	0.95	0.97	0.89	0.98	1.03	0.98	0.97	0.89	0.87	0.63	0.85
	QPN	0.13	0.91	0.90	0.98	0.95	0.99	0.88	0.62	0.97	0.98	0.60	0.75	0.75	0.80	0.04	0.84	0.50	0.78	0.78	0.97	0.80	0.93	0.88	0.60	0.52	0.69
99th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.77	1.01	1.03	1.02	0.99	1.00	1.00	1.00	1.00	1.00	1.00	0.98	1.00	0.98	0.66	1.00	1.00	1.00	1.01	1.00	1.00	1.00	0.96	1.00	1.00	0.97
	QTCW	0.77	1.02	1.04	1.02	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.67	1.00	1.00	1.00	1.01	1.00	1.00	1.00	0.96	1.00	1.00	0.97
	QPN	0.77	0.95	1.01	1.02	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.66	0.99	1.00	1.00	0.73	1.00	0.85	0.99	0.95	1.00	1.00	0.92

Table 6.4 Hit rate of Q-table cache only method.

	3D 128Gb										
	TPCC	EXCH24	RA*	DTR*	MSNSFS*	RBESQL	MSNSFS	Financial1	oltp	RBESQL*	home2
Hit rate [%]	89	50	31	43	48	45	32	88	85	48	75

6.3.4 Q-value Prediction Network Effects on Latency

In this subsection, we report how the size of QP Net is determined. We also evaluate alternative designs related with QP Net, i.e., QP Net only solution and actor-critic method [11].

• **QP Net size:** The QP Net size needs to be minimized to reduce the additional cost of runtime and memory due to QP Net execution. We performed a sensitivity analysis to determine the appropriate QP Net size by varying the number of neurons in the two hidden layers. Figure 6.7 shows the results (the average latency of the total workloads at the 99.9999^{th} percentile) for different sizes of the QP Net in the 3D 512Gb and 3D 128Gb flash memories. The 3D 512Gb flash memory has the best (lowest) latency with 50 neurons per hidden layer, and the 3D 128Gb flash memory has the best (lowest) latency with 100 neurons per hidden layer.

The fact that the smaller capacity flash memory requires the larger QP Net can be analyzed as follows. In the case of the 3D 128Gb flash memory, because of the smaller capacity, more GCs are executed and $1.38\times$ more states are created than in the 3D 512Gb flash memory. Thus, in order to account for more states, a larger QP Net is needed.

Though we assume that the QP Net size is determined in design time (50/100 neurons in each hidden layer for 512Gb/128Gb flash memories) when the type of flash memory is determined, we do not exclude the pos-

sibility of dynamically adjusting QP Net size during runtime depending on the available resource, e.g., storage capacity change due to aging, and the application behavior, e.g., a large number of active states incurring low Q-table cache hits. Investigating the possibility of further reduction in latency in such cases will be future work.

• **Pre-training effects:** We compared two pre-training options: random and realistic traces. After the pre-training, we used, as the initial configuration, the pre-trained QP Net and zero-initialized QTC and evaluated our integrated solution with the realistic traces. In the case of random traces, we used 10,000 randomly generated requests to train both Q-table cache and QP Net. In the case of pre-training with realistic traces, we utilized seven realistic traces (*home2*, webmail+online*, home1*, RBESQL*, RA*, MSNSFS*,* and *webmail**). We performed pre-training and evaluation in 7-fold cross-validation (i.e., leave-one-out). For instance, we trained both Q-table cache and QP Net with all traces except for *home2** and evaluated the Q-table cache and pre-trained QP Net with *home2**.

Table 6.5 compares the normalized latency of the two pre-training options for the 99.9999^{th} percentile. On average, the pre-training with random traces gives 21% better (lower) latency than that with realistic traces.

We analyze that the pre-training with random traces can give much smaller Q-value prediction error than that with realistic traces as shown

in Table 6.6. The table compares the Q-value prediction error of QP Net. We define the error as (Q(QTC)-Q(QPNet))/Q(QTC) where Q(QTC) and Q(QPNet) represent the Q-value of Q-table cache and that of QP Net for the same input state. We calculate the error with the Q-values after each update in Q-table cache and QP Net. As the table shows, the pre-training with random traces gives much smaller error than that with realistic traces. The large error of realistic traces could result from overfitting to the realistic traces. In our future work, we will further investigate how to exploit realistic traces as much as possible while benefiting from random traces.

We also evaluated the case that QP Net weights are randomly initialized without pre-training (called **No Pre-Training**). As the tables show, the case of no pre-training gives smaller latency and prediction error than the pre-training with realistic traces while being inferior to the proposed pre-training with random traces. This case also shows that the pre-training with realistic traces may suffer from overfitting when comparing the prediction error between no pre-training and pre-training with realistic traces.

• **Latency of the QP Net without the Q-table cache:** Our integrated solution uses both the Q-table cache and the QP Net. However, we can think of a solution that uses only the QP Net without the Q-table cache because the QP Net can give Q-values on the given state. We implemented a solution using only the QP Net without the Q-table cache. We

can consider the performance difference between our integrated solution and this QP Net only one as the contribution of Q-table cache in our integrated solution. Note that the QP Net continues to be trained during runtime in both solutions.

Table 6.7 compares the latency of the integrated solution (**QPN**) and the QP Net only one (**NET**). The table shows that the integrated solution (QPN) gives better (lower) average latency than the QP Net only one (NET). To be specific, at 99.9999^{th} percentile, the integrated solution offers the average latency of $0.75\times$ and $0.63\times$ for the 3D 512Gb and 3D 128Gb flash memories, respectively. On the other hand, the QP Net only one shows the latency of $1.10\times$ and $1.01\times$, for the 3D 512Gb and 3D 128Gb, respectively.

The QP Net only solution is not equipped with the Q-table cache. Thus, in order to achieve a similar performance, it might require a larger network than the integrated solution. Thus, we conducted the experiments by varying the network size in the QP Net only solution.

Figure 6.8 shows the average normalized latency of the QP Net only solution for various QP Net sizes (number of neurons in each hidden layer) at 99.9999^{th} percentile. The results show that the integrated solution (**QPN**) gives better (lower) average latency than the QP Net only one even under various QP Net sizes.

Figure 6.8 proves the benefit of our integrated solution where the Q-table cache is specialized to exploit the short-term history while the QP

Net learns the long-term history in order to provide good initial Q-values to the Q-table cache thereby enabling better Q-learning and, finally, lower latency.

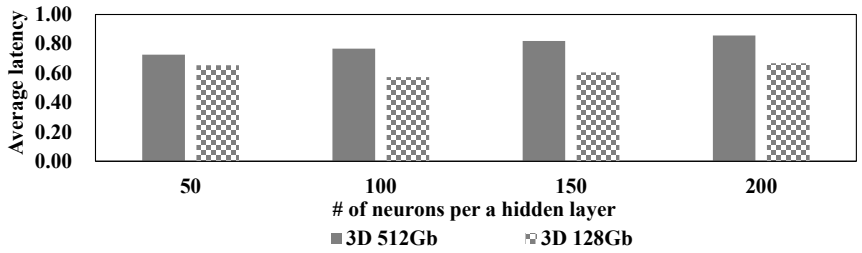


Figure 6.7 Sensitivity analysis: latency vs. QP Net size.

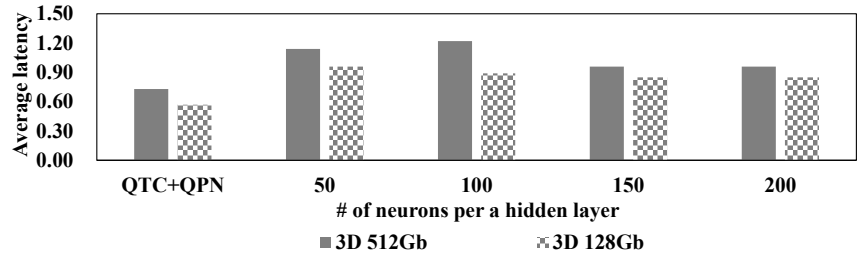


Figure 6.8 Latency comparison of QP Net without Q-table cache for various network sizes at 99.9999^{th} percentile (normalized to the baseline).

Table 6.5 Pre-training comparison.

	home1*	home2*	web+online*	webmail*	MSNSFS*	RBESQL*	RA*	AVERAGE
No Pre-Training	1.00	1.00	0.82	0.99	0.97	0.84	0.69	0.90
Random	0.75	1.00	0.81	0.65	0.81	0.92	0.68	0.80
Realistic	1.00	1.00	0.86	0.99	0.81	1.16	1.24	1.01

Table 6.6 Q-value prediction error comparison.

	home1*	home2*	web+online*	webmail*	MSNSFS*	RBESQL*	RA*	AVERAGE
No Pre-Training	0.296	0.206	0.325	0.420	0.123	0.139	0.144	0.236
Random	0.057	0.012	0.046	0.085	0.086	0.119	0.067	0.067
Realistic	0.315	0.243	0.357	0.399	0.087	0.098	0.318	0.260

Table 6.7 Latency comparison of QP Net without Q-table cache (normalized to the baseline).

Percentile		(a) 3D 512Gb (50 neurons per hidden layer)														(b) 3D 128Gb (100 neurons per hidden layer)											
		TPCC	EXCH24	MSNSFS	MSNSFS*	oltp	RBESQL	DTR*	webmail*	RA*	Financial1	RBESQL*	home1*	web+online*	AVERAGE	TPCC	EXCH24	RA*	DTR*	MSNSFS*	RBESQL	MSNSFS	Financial1	oltp	RBESQL*	home2	AVERAGE
99.	QPN	0.15	0.82	0.93	0.87	1.00	0.88	0.81	0.61	0.47	0.78	0.86	0.78	0.75	0.75	0.02	0.80	0.31	0.79	0.62	0.84	0.88	0.65	0.88	0.80	0.33	0.63
9999^{th}	NET	0.60	0.91	1.18	0.83	1.52	0.95	0.93	0.62	0.59	3.27	0.85	1.20	0.89	1.10	0.19	0.89	0.92	1.09	1.98	1.02	1.28	0.76	0.91	0.89	1.14	1.01

• **Latency of applying the actor-critic method:** As mentioned earlier, this study was also motivated by the actor-critic method [30]. The actor-critic method approximates the probability of action and value functions. We implemented a solution using the actor-critic method to compare the performance with our integrated solution. Table 6.9 shows the architecture of the actor-critic network we implemented.

Table 6.8 compares the latency of our integrated solution (**QPN**) and the one using actor-critic method (**AC**) at 99.9999^{th} percentile. The table shows that our solution gives lower (better) average latency than the actor-critic one for both the 3D 512Gb and the 3D 128Gb flash memories. To be specific, our solution gives the average latency of $0.75\times$ and $0.63\times$ for the 3D 512Gb and the 3D 128Gb flash memories, respectively, while the actor-critic one offers the average latency of $0.85\times$ and $0.74\times$ for two memory types.

We also implemented a solution using the asynchronous advantage actor-critic method (A3C) to compare the performance. Fig. 6.9 shows a high level architecture of A3C. As shown in the figure, total four agents (one global agent and three local agents) are employed. A global agent works as an real-SSD and local agents work as virtual-SSD to collect experiences under the independent environment. Table 6.9 shows the architecture of the actor-critic network we implemented for each agent. Unlike the original A3C method, we use the online learning method to apply A3C to real time SSDs. Note that this A3C method cannot apply

to real SSD, because real SSD can not virtualize their environment.

Table 6.8 compares the latency of our integrated solution (**QPN**) and the one using A3C method (**A3C**) at 99.9999^{th} percentile. The table shows that our solution gives lower (better) average latency than the A3C one for both the 3D 512Gb and the 3D 128Gb flash memories. To be specific, our solution gives the average latency of $0.75\times$ and $0.63\times$ for the 3D 512Gb and the 3D 128Gb flash memories, respectively, while the A3C one offers the average latency of $0.87\times$ and $0.76\times$ for two memory types. The results of actor-critic method shows slightly better latency reduction than A3C.

Actor-critic one has the advantage of using continuous values as states instead of binned states. However, network learning is slow. In addition, the predicted approximate values are directly used to determine the action probability and the target value. Therefore, the effect of approximation error can be large. In terms of implementation cost, as shown in Table 6.9, the actor critic method requires larger neural networks for actor and critic networks than our QP Net.

On the other hand, our integrated solution learns the policy faster than the actor-critic one because ours runs on a small Q-table cache. QP Net learning can be slow like an actor-critic one. However, QP Net plays a role of predicting the initial value of Q-table cache. Therefore, the effect of the approximation error due to slow learning can be smaller than that of the actor-critic one.

In cases of *oltp* for both flash memory types and *RBESQL** for the 3D 128Gb flash memory, the actor-critic one shows slightly better performance as shown in Table 6.8. We consider that it is due to the benefit of using continuous values in the actor-critic method. As Table 6.10 shows, *oltp* has a high hit rate of the Q-table cache. In case of a high hit rate, the influence of QP Net is small. In such a case, a better binning or the usage of continuous values of the actor-critic method could lead to a better solution.

In the case of *RBESQL**, Table 6.10 shows the hit rate of the Q-table cache is low. That is, the number of states used in *RBESQL** is very large and the states may vary frequently, which makes it difficult to take full advantage of the Q-table cache [10]. The low hit rate of Q-table cache can have an adverse impact on latency due to frequent evictions, i.e., information loss, though the QP Net provides good initial Q-values. Table 6.10 also shows the workload behavior of *RBESQL** in terms of average request size and standard deviation which are larger than other workloads, confirming the characteristics of a large number of states. In such a case, the actor-critic method, which utilizes continuous values without suffering from information loss due to Q-table cache eviction, could outperform the Q-table cache-based integrated solution.

Note that, by introducing the QP Net, the action choice and the state behavior change thereby giving different hit rates in Tables 6.4 and 6.10. Specifically, compared with the QTC only case, the QP Net can change

the selected action due to the better Q-learning. Different actions can change the state behavior. For instance, if more partial GC operations are taken in the QP Net case, then the number of free blocks will tend to increase, which will frequently generate the states having more free blocks than in the QTC only case.

Table 6.8 Latency comparison of actor-critic and A3C methods (normalized to the baseline).

Percentile		(a) 3D 512Gb (50 neurons per hidden layer)														(b) 3D 128Gb (100 neurons per hidden layer)										
		TPCC	EXCH24	MSNSFS	MSNSFS*	oltp	RBESQL	DTR*	webmail*	RA*	Financial1	RBESQL*	home1*	web+online*	AVERAGE	TPCC	EXCH24	RA*	DTR*	MSNSFS*	RBESQL	MSNSFS	Financial1	oltp	RBESQL*	home2	AVERAGE
99. 9999th	QPN	0.15	0.82	0.93	0.87	1.00	0.88	0.81	0.61	0.47	0.78	0.86	0.78	0.75	0.75	0.02	0.80	0.31	0.79	0.62	0.84	0.88	0.65	0.88	0.80	0.33	0.63
	AC	0.18	0.88	1.21	0.83	0.97	0.98	0.88	0.71	0.76	0.99	0.99	0.88	0.75	0.85	0.08	0.84	0.74	0.83	0.75	0.99	1.09	0.90	0.86	0.73	0.35	0.74
	A3C	0.18	0.90	1.31	0.83	0.99	0.97	0.89	0.80	0.78	0.98	0.99	0.90	0.82	0.87	0.08	0.87	0.80	0.85	0.74	0.99	1.11	0.90	0.87	0.77	0.35	0.76

Table 6.9 Actor-critic network architecture.

	Actor	**Critic**
Number of inputs (states)	17	17
Number of hidden layers	2	3
Number of nodes / hidden layer	100	500
Number of outputs	3	1

Table 6.10 Workload information (hit rate and state counts are from integrated solution).

	3D 128Gb						
	home2	MSNSFS*	RBESQL*	oltp*	TPCC	MSNSFS	RBESQL
Hit rate [%]	80	50	55	93	97	37	46
# of states	45559	337923	183545	6346	4293	435194	203081
Avg. req. sectors	9.40	21.67	57.85	4.46	17.16	21.67	57.85
SD of req. sectors	18.11	45.75	162.75	15.5	8.65	45.75	162.75

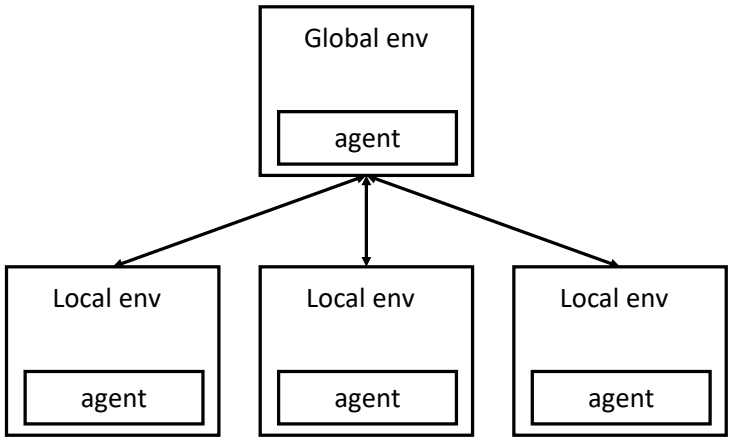


Figure 6.9 Architecture of asynchronous advantage actor-critic method.

6.3.5 Q-table Cache Analysis

• **Replacement policy of Q-table cache:** Our proposed solution employed an LRU policy as the Q-table cache replacement policy. In order to evaluate the possibilities of different replacement policies, we compared the performance of original Q-table cache method with LRU, LIRS [43] and ARC [44] under the same constraint of memory usage, i.e., 1,600 bytes for the memory cost of 100-entry QTC with LRU. Both LIRS and ARC consider frequency as well as recency to compensate for the disadvantages of LRU. Thus, both policies require storing meta data, i.e., the state information of evicted entries, which gives fewer entries in the QTC under the same memory cost. In our experiments, in order to meet the memory cost of 1,600 bytes, we used 80-entry QTC for LIRS and ARC, respectively.[7] Our experiments (the corresponding results of which are omitted due to page limit) show that, LRU outperforms both of LIRS and ARC in the two flash memory types. It is mainly because, under the same constraint of memory cost, LRU gives more QTC entries than LIRS and ARC. As explained earlier, LIRS and ARC need additional memory area for storing meta data, which reduces the number of QTC entries under the same memory constraint.

• **Large Q-table cache:** Our proposed integrated solution requires

[7] In our experiments, both LIRS and ARC keep the state information of 80 evicted entries. Each entry of QTC has 16 bytes (=3 actions $\times$ 4 bytes/action + 4 bytes/state information). The state information of evicted entry has 4 bytes. Thus, for each of LIRS and ARC, the 80-entry QTC meets the memory cost of 1,600 bytes = 80 entries $\times$ (16 bytes/entries + 4 bytes/evicted entry).

additional memory cost of the QP Net in the original Q-table cache method. For a fair evaluation, we need to compare latency under the same memory cost when increasing the size of the original Q-table cache to use the same amount of memory as the integrated solution. Table 6.12 shows the memory cost of the integrated solution (details of memory cost are discussed later). As shown in Table 6.12, the integrated solution requires 25.96KB and 88.68KB for 50 and 100 neurons per hidden layer, respectively. One Q-table cache entry requires 16B (described in Section 6.2.2). Thus, for iso-memory cost comparison, we utilize 1,107 and 3,783 Q-table cache entries for 3D 512Gb and 3D 128Gb flash memories, respectively.

Table 6.11 compares the latency of our integrated solution (**QPN**) and that of the large Q-table cache only solution (**LQTC**) at 99.9999^{th} percentile. The table shows that our proposed solution offers lower (better) average latency than the large Q-table cache for both the 3D 512Gb and 3D 128Gb flash memories. The large Q-table cache can manage more state information than the original Q-table cache. However, the learned information is still lost due to state eviction. Especially, a larger Q-table cache can suffer from wrong action choices in immature states. On the other hand, the proposed solution tries to avoid such immature states by predicting a better initial value of the Q-table cache.

Table 6.11 Latency comparison of large Q-table cache (normalized to the baseline).

Percentile		(a) 3D 512Gb (50 neurons per hidden layer)														(b) 3D 128Gb (100 neurons per hidden layer)									
		TPCC	EXCH24	MSNSFS	MSNSFS*	oltp	RBESQL	DTR*	webmail*	RA*	Financial1	RBESQL*	home1*	web+online*	AVERAGE	TPCC	EXCH24	RA*	DTR*	MSNSFS*	RBESQL	MSNSFS	Financial1	oltp	RBESQL*	home2	AVERAGE
99.9999th	QPN	0.15	0.82	0.93	0.87	1.00	0.88	0.81	0.61	0.47	0.78	0.86	0.78	0.75	0.75	0.02	0.80	0.31	0.79	0.62	0.84	0.88	0.65	0.88	0.80	0.33	0.63
	QTC	0.20	0.98	1.03	1.02	1.04	0.95	0.93	0.93	0.70	1.00	0.92	1.00	0.88	0.89	0.12	0.87	0.68	0.86	1.01	0.97	1.01	0.96	0.92	0.85	0.34	0.78
	LQTC	0.31	0.95	0.93	1.00	1.02	0.94	0.89	0.81	0.62	0.90	0.89	0.90	0.83	0.85	0.55	0.86	0.36	0.77	1.06	1.02	1.04	0.71	0.90	0.97	0.33	0.78
	LQ20	0.31	1.00	0.99	1.01	1.04	0.98	0.91	0.93	0.90	1.25	1.13	0.82	0.82	0.93	0.46	0.88	0.48	0.97	1.00	0.98	1.14	0.71	0.90	1.02	0.53	0.82
	LQ40	0.31	1.00	1.08	1.01	1.04	0.98	0.92	0.93	0.90	1.29	1.17	0.82	0.99	0.96	0.46	0.88	0.56	1.06	1.01	1.05	1.18	0.71	0.90	1.02	0.63	0.86

6.3.6 Immature State Analysis

We observed that in some workloads (*TPCC* in 3D 512Gb flash memory and *TPCC, MSNSFS, MSNSFS*, RBESQL,* and *RBESQL** in 3D 128Gb flash memory), the original small Q-table cache **(QTC)** outperforms the large Q-table cache. As shown in Table 6.10, *MSNSFS, MSNSFS*, RBESQL,* and *RBESQL** show a relatively large number of states and low hit rates. Due to the large number of states and the low locality of the states, the large Q-table cache also suffers from low hit rates, e.g., 56% (*MSNSFS*), 64% (*MSNSFS**), 62% (*RBESQL*) and 68% (*RBESQL**), respectively.

As will be explained below, in such a case, the problem of immature states is significant thereby degrading the quality of the large Q-table cache. On the other hand, *TPCC* shows a high hit rate in the original small Q-table cache. In such a case, increasing the size of Q-table cache could increase the number of immature states worsening the quality of Q-table cache.

In order to analyze the amount of immature states, as shown in Figure 6.10, we obtained the histogram of the access frequency of states which remain on the large Q-table cache after workload runs.[8] In the figure, the x-axis represents the access frequency and the y-axis the number of states corresponding to the access frequency range. As shown in the figure, the workloads (*TPCC, MSNSFS, MSNSFS*, RBESQL,* and *RBESQL**),

[8]The figure shows a snapshot of Q-table cache at the end of trace run. We use this as a representative state of system run.

explained above, have a large number of states with small (i.e., less than ten) access frequency. We consider such states as immature states[9] which degrade the quality of large Q-table cache.

The original small Q-table cache mitigates the negative impact of immature states in two ways. First, the small Q-table cache tends to have a smaller number of immature states. Second, in case of cache miss (which often corresponds to the access to immature states in the large Q-table cache), the small Q-table cache takes a conservative action choice, i.e., action 0 (No GC) which does not give an immediate latency increase while the large Q-table cache can select an inappropriate action from an immature state. Note also that the tail latency (i.e., 99.9999^{th} percentile) can often be affected by a single inadequate action selection due to its infrequent occurrence.

On the contrary, *RA** and *Financial1* show better (low) latency in the large Q-table cache than in the small cache. As Figure 6.10 shows, these workloads have a relatively small number of immature states. Therefore, the possibility of selecting inappropriate actions could be lower.

Another observation in some workloads (*MSNSFS, RBESQL, MSNSFS** and *RBESQL** in both 3D 512Gb and 3D128Gb flash memory) is that as shown in Table 6.11, their latency on LQTC is worse than QTC only in 3D128Gb, but not in 3D512Gb. As shown in the original Q-table

[9]The definition of state maturity is difficult to obtain. Thus, our classification of immature states could be subjective.

cache solution [10], the Q-table cache size that shows good performance depends on the characteristics of the workload. *MSNSFS* and *RBESQL* show the smallest latency in a Q-table cache with 2,000 entries, and the latency increases as the size of the Q-table cache gets larger. If the size of the Q-table cache is too small, there is a high probability of losing the learned information. On the other hand, if the size of the Q-table cache is too large, the influence of immature states can become more significant. As mentioned earlier, we used Q-table cache with 1,107 and 3,783 entries for 3D512Gb and 3D128Gb, respectively. In the case of 3D512Gb (1,107 Q-table cache entries), latency could be reduced as the possibility of losing the learned information gets reduced. On the other hand, in the case of 3D128Gb (3,783 Q-table cache entries), the latency tends to increase due to the negative effect of a large number of immature states.

In order to resolve the problem of immature states, we also evaluated a simple heuristic which applies thresholding based on he count of state visits. If the count is smaller than a threshold, we select the conservative choice, action 0 while continuing to train the Q-table and QP Net. Table 6.11 gives latency comparison. **LQ20** and **LQ40** represent the cases of large Q-tables with the threshold of 20 and 40, respectively. As the table shows, the heuristics gives similar results to that of the baseline large Q-table (LQTC). We expected that the conservative choice of action may be useful to avoid long latency due to wrong choices from immature states. However, the heuristics turns out to suffer from immature states

because only the Q-values of action 0 are trained when the counts are small thereby rendering the states of the other actions immature ones. We expect there is a potential of further improvements by judiciously handling immature states especially when a large QTC is available, which is left for future work.

6.3.7 Miscellaneous Analysis

• **Reward assignment options for action 0:** As mentioned in Section 6.2.2, it is important to deal with negative reward in case of action 0. We compared two reward assignment options: negative and zero reward for action 0 on our integrated solution. Table 6.13 compares average latency (normalized to the baseline) at 99.9999^{th} percentile. As the table shows, zero reward option gives by 11% and 12% lower average latency on the 3D 512Gb and 3D 128Gb flash memories, respectively. This shows the effectiveness of zero reward assignment in the problem of latency increase in action 0.

• **Computation and memory overhead:** We measured the computation overhead of the QP Net, i.e., the training and prediction time of the QP Net (prediction and training) and that of the Q-table cache access (search and insert time of the Q-table cache) on the ZED board [45] with a Cortex A9 processor. The Cortex R5 processor [46] is mainly used for SSD products [47, 48]. Therefore, the results measured on the Cortex A9 are converted to those of Cortex R5 using the performance information

of DMIPS / MHz [46] provided by ARM. Note that, as we mentioned earlier, our integrated solution does not trigger the partial GC in case of read requests.

Table 6.14 shows the training and prediction runtime of the QP Net, both of which are smaller than a typical program time (tPROG, $\sim$1ms) in both cases of 50 and 100 neurons per hidden layer. Therefore, the computation overhead of QP Net training/prediction can be considered negligible since it can be hidden by the write latency to the flash memory. In addition, the computation overhead can be further reduced in case that the QP Net can run on a hardware accelerator equipped on the SSD [49–51].

The Table 6.15 shows the search and insert time of the Q-table cache with 100 entries. In case of insert, the runtime covers a search of 100 entries and eviction/insertion. As both tables show, the computation overhead of our integrated solution (Q-table cache and QP Net) can be considered negligible since it can be hidden by the write latency to the flash memory.

Table 6.12 shows the memory overhead of our integrated solution. As the table shows, QP Net includes 6,153 and 22,303 parameters (each 4 bytes) for 50 and 100 neurons per hidden layer, respectively. The Q-table cache size with 100 entries for three actions is 1.56KB (described in Section 6.2.2). Thus, the total memory overhead of both Q-table cache and QP Net is 25.96KB/88.68KB for QP Net with 50/100 neurons per

hidden layer. The memory cost is negligible considering the fact that the DRAM buffer size of SSD is much larger, typically 1GB for 1TB capacity [52–54].

• **Erase count:** Tables 6.16 and 6.17 compare the erase count for the 3D 512Gb and 3D 128Gb flash memories. The results are normalized to the baseline. In both cases, the erase count of the proposed integrated solution is similar to the baseline.

• **Block size effects:** In order to evaluate the effects of block size, we varied the block size (# pages/block) from 384 to 1,536 and the number of blocks to keep the same capacity for 3D 512Gb flash memory (the original block size of which is 768), and measured latency at 99.9999^{th} percentile on the baseline and proposed solution. Our experiments (the corresponding results of which are omitted due to page limit) show that both of baseline and QPN increase the latency as block size increases. However, QPN is less correlated with that of block size than the baseline. It is because QPN basically offers smaller latency than the baseline. Thus, it will help to reduce latency increase with increasing block size.

Table 6.12 Memory cost.

	# of parameters	
	50 neurons	100 neurons
Input layer	900	1800
Hidden layer	2550	10100
Hidden layer	2550	10100
Output layer	153	303
Total	6153	22303
QP Net size [KB]	24.04	87.12
Q-table cache size [KB]	1.56	1.56
Total mem size [KB]	25.96	88.68

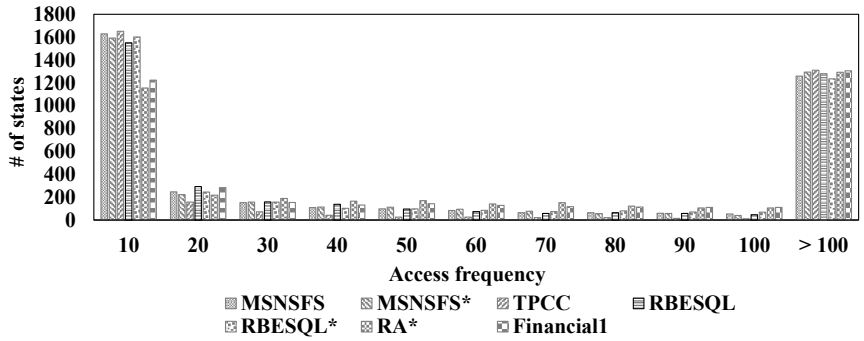


Figure 6.10 Number of states in Q-table cache for each access frequency in 3D 128Gb (after running the workloads).

Table 6.13 Average latency comparison between negative and zero reward for action 0.

Reward	Average latency at 99.9999th percentile	
	3D 512Gb	3D 128Gb
Negative	0.86	0.75
Zero	0.75	0.63

Table 6.14 Computation overhead [μs].

		# of neurons per a hidden layer			
		50	100	150	200
Evaluation	Measured on Cortex A9	35.15	105.55	215.15	368.30
	Converted for Cortex R5	52.72	158.32	322.72	552.45
Training	Measured on Cortex A9	81.80	290.80	643.25	1119.50
	Converted for Cortex R5	122.70	436.20	964.88	1679.25

Table 6.15 Computation overhead of Q-table Cache [μs].

Search	Measured on Cortex A9	2.90
	Converted for Cortex R5	4.34
Insert	Measured on Cortex A9	12.30
(Search+Erase+Insert)	Converted for Cortex R5	18.41

Table 6.16 Erase count for 3D 512Gb flash memory.

	TPCC	EXCH24	MSNSFS	MSNSFS*	oltp	RBESQL	DTR*	webmail*	RA*	Financial1	RBESQL*	home1*	web+online*	AVERAGE
Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
QTC	0.99	1.00	1.03	1.02	1.00	0.99	0.99	1.00	0.98	1.00	1.00	1.00	1.00	1.00
QPN	0.98	0.99	1.02	0.99	0.91	0.99	0.99	1.00	0.99	1.00	1.00	1.00	1.00	0.99

Table 6.17 Erase count for 3D 128Gb flash memory.

	TPCC	EXCH24	RA*	DTR*	MSNSFS*	RBESQL	MSNSFS	Financial1	oltp	RBESQL*	home2	AVERAGE
Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
QTC	0.99	1.00	0.99	1.00	1.00	1.00	1.02	1.00	1.00	1.00	1.03	1.00
QPN	0.93	1.00	1.00	1.01	0.97	1.00	0.99	0.97	1.00	1.00	1.03	0.99

6.3.8 Multi Channel Analysis

The evaluations discussed so far were conducted in a single channel SSD environment. To increase the reality and robustness of the proposed solution, we performed additional experiments in a 4-channel SSD environment.

In addition to the workload used in the previous experiment, five additional workloads were utilized (*randomtransaction, readwhilewriting, overwrite, filluniquerandom*, and *readrandomwriterandom*). These were extracted using RocksDB [55]. And three additional real world workloads (*VDI0222LUN1, VDI0223LUN2*, and *VDI0224LUN3*) [56] were utilized.

Table 6.18 shows latency comparison of 3D 512 GB flash memory in the 4-channel environment. Few workloads offer latency reduction for both of our proposed solutions. This is a consequence of the capacity of SSD having quadrupled with the usage of 4 channels. Therefore, the incidence of GC decreases, and GC induced latency delay is not significant.

We reduced the number of blocks of flash memory by a factor of four to evaluate the proposed solution in the 4-channel environment, while maintaining the same SSD capacity as the previous experiment. All subsequent experiments were performed using this configuration.

• **Latency comparison at four channel configuration:** Table 6.19 shows latency comparison in 4 channel configuration on 3D512Gb flash

Table 6.18 Latency comparison without block size reduction for 3D 512Gb flash memory (4CH).

Percentile		TPCC	EXCH24	MSNSFS	MSNSFS*	oltp	RBESQL	DTR*	webmail*	RA*	Financial1	RBESQL*	home1*	web+online*	randtran	rdwhilewr	overwrite	fillunirand	rdrandwrran	V22LUN1	V23LUN2	V24LUN3	AVERAGE
99.9999th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.82	1.00	1.00	1.00	1.00	1.00	0.71	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.78	0.79	0.59	0.94
	QPN	0.80	1.00	1.00	1.00	1.00	1.00	0.69	1.00	1.00	0.94	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.55	0.57	0.59	0.91
99.99th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.82	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00	0.79	0.85	0.83	0.97
	QPN	0.77	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00	0.78	0.81	0.83	0.96
99th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.84	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99
	QPN	0.83	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99

memory. Both of our proposed methods (QTC and QPN in the table) show better (lower) average latency than the baseline. QTC gives latency reductions by 0.89× at 99.9999^{th} percentile, 0.90× at 99.99^{th} and 0.97× at 99^{th}. QPN offers further reductions by 0.83× at 99.9999^{th}, 0.86× at 99.99^{th} and 0.95× at 99^{th}.

The baseline (Base in the table) uses a small number of states to learn policy. However, QTC exploits a much larger number of fine-grained states and maintains key states among them. Thus, it offers smaller latency than the baseline.

Our integrated solution of Q-table cache and QP Net (QPN in the table) gives further latency reductions by training QP Net during runtime to provide better initialization of Q-table cache than the zero initialization of the original Q-table cache, which finally contributes to better action selection. Note that the latency improvement of QPN comes from better Q-value initialization since both the QTC and QPN utilize the same number of candidate states.

Table 6.20 also compares the latency on the 3D 128Gb flash memory. Our methods show better (lower) average latency than the baseline by 0.86×(QTC)/0.80×(QPN) at the 99.9999^{th} percentile, 0.89×(QTC)/ 0.85×(QPN) at 99.99^{th}, and 0.99×(QTC)/0.97×(QPN) at 99^{th}. That is, the QP Net gives additional 4-6% reductions to the original Q-table cache [10] in the two types of flash memory.

• **Applying Erase/Program suspension:** Modern flash memory pro-

Table 6.19 Latency comparison for 3D 512Gb flash memory (4CH).

Percentile		TPCC	EXCH24	MSNSFS	MSNSFS*	oltp	RBESQL	DTR*	webmail*	RA*	Financial1	RBESQL*	home1*	web+online*	randtran	rdwhilewr	overwrite	fillunirand	rdrandwrran	V22LUN1	V23LUN2	V24LUN3	AVERAGE
99.9999th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.34	0.92	0.90	1.00	1.00	1.00	0.92	0.86	0.94	1.00	1.00	1.00	0.89	0.99	0.90	0.61	0.93	1.00	0.81	0.76	0.82	0.89
	QPN	0.30	0.82	0.81	1.00	1.00	0.91	0.84	0.80	0.83	0.98	0.96	0.93	0.82	0.99	0.90	0.58	0.86	1.00	0.62	0.65	0.81	0.83
99.99th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.43	0.98	0.93	1.00	0.93	1.00	1.00	0.98	1.01	1.00	0.96	1.00	0.86	0.99	0.82	0.61	0.92	1.00	0.84	0.80	0.81	0.90
	QPN	0.38	0.85	0.85	1.00	0.93	0.96	0.97	0.98	0.96	1.00	0.91	0.94	0.84	0.99	0.82	0.59	0.92	1.00	0.64	0.68	0.81	0.86
99th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.87	0.88	0.94	0.95	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.93	1.01	0.84	1.02	0.96	1.00	0.99	1.00	0.97
	QPN	0.87	0.72	0.87	0.95	0.98	1.00	1.00	1.00	1.00	0.99	1.00	1.00	1.00	0.93	1.00	0.84	0.98	0.94	1.00	0.95	1.00	0.95

Table 6.20 Latency comparison for 3D 128Gb flash memory (4CH).

Percentile		TPCC	EXCH24	RA*	DTR*	MSNSFS*	RBESQL	MSNSFS	Financial1	oltp	RBESQL*	home2	randtran	rdwhilewr	overwrite	fillunirand	rdrandwrran	V22LUN1	V23LUN2	V24LUN3	AVERAGE
99. 9999th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.37	0.93	0.83	0.93	1.00	1.00	0.98	1.00	1.00	0.91	0.75	1.00	0.95	0.58	0.90	1.00	0.81	0.72	0.79	0.87
	QPN	0.32	0.83	0.75	0.90	0.96	0.91	0.87	0.89	1.00	0.86	0.67	1.00	0.95	0.53	0.86	1.00	0.58	0.68	0.71	0.80
99. 99th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.53	0.97	0.88	0.95	1.00	0.99	1.01	1.00	1.00	0.98	0.71	1.00	0.88	0.65	0.93	0.98	0.84	0.86	0.75	0.89
	QPN	0.50	0.93	0.81	0.94	1.00	0.92	0.97	0.90	1.00	0.93	0.68	1.00	0.88	0.58	0.92	1.00	0.63	0.82	0.70	0.85
99th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	1.00	1.01	1.00	1.00	1.00	1.00	1.00	1.00	0.85	1.00	1.00	0.97	1.02	0.83	1.00	1.00	0.98	1.00	0.99	0.98
	QPN	0.98	0.94	1.00	1.00	0.92	1.00	1.00	0.99	0.84	1.00	1.00	0.99	1.02	0.80	1.00	1.00	0.97	1.00	0.97	0.97

vides a suspension function to achieve fast read latency [57]. When a read request is received, a program or an erase operation, which requires relatively longer latency than a read operation, is stopped by this scheme. The suspended program or erase operation is resumed after the completion of the operation (i.e., read operation) that requires fast latency.

Tables 6.21 and 6.22 compare the normalized latency of the original our solution and the case of applying suspension scheme for 3D512G and 3D128G flash memories. We implemented the suspension scheme to both of our solutions (QTCSUS and QPNSUS in the table). The tables show that the case of applying suspension scheme gives similar average latency than our original solution for both flash memories. As shown in the tables, both our original solution and the case of applying suspension give similar average latency reductions which are 0.89×(QTC)/0.87× (QTCSUS), 0.83×(QPN)/0.80×(QPNSUS) on the 3D512G flash memory. The results of the 3D 128G flash memory gives average latency reductions which are 0.86×(QTC)/0.84×(QTCSUS), 0.80×(QPN)/0.79× (QPNSUS).

• **Demand-based Q-table cache:** As mentioned earlier, our proposed Q-table cache solution uses a small size of 100 entries. As Q-table cache has the limitation of losing learned information, we can achieve better performance if we increase the size of the Q-table cache significantly; in this case, we use 500,000 entries. However, with substantial increase in the Q-table cache size, it becomes difficult to store in the memory (static

Table 6.21 Latency comparison of applying suspension scheme for 3D 512Gb flash memory (4CH).

Percentile		TPCC	EXCH24	MSNSFS	MSNSFS*	oltp	RBESQL	DTR*	webmail*	RA*	Financial1	RBESQL*	home1*	web+online*	randtran	rdwhilewr	overwrite	fillunirand	rdrandwrran	V22LUN1	V23LUN2	V24LUN3	AVERAGE
99.9999th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.34	0.92	0.90	1.00	1.00	1.00	0.92	0.86	0.94	1.00	1.00	1.00	0.89	0.99	0.90	0.61	0.93	1.00	0.81	0.76	0.82	0.89
	QTCSUS	0.32	0.90	0.90	0.98	1.00	0.96	0.89	0.83	0.93	0.98	0.97	1.00	0.87	0.99	0.90	0.61	0.91	1.00	0.80	0.74	0.81	0.87
	QPN	0.30	0.82	0.81	1.00	1.00	0.91	0.84	0.80	0.83	1.00	0.96	0.93	0.82	0.99	0.90	0.58	0.86	1.00	0.62	0.65	0.81	0.83
	QPNSUS	0.28	0.82	0.75	1.00	1.00	0.90	0.78	0.80	0.72	0.97	0.84	0.88	0.82	0.99	0.90	0.56	0.85	1.00	0.58	0.64	0.79	0.80

Table 6.22 Latency comparison of applying suspension scheme for 3D 128Gb flash memory (4CH).

Percentile		TPCC	EXCH24	RA*	DTR*	MSNSFS*	RBESQL	MSNSFS	Financial1	oltp	RBESQL*	home2	randtran	rdwhilewr	overwrite	fillunirand	rdrandwrran	V22LUN1	V23LUN2	V24LUN3	AVERAGE
99.9999th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.37	0.93	0.83	0.93	1.00	1.00	0.98	1.00	1.00	0.91	0.75	1.00	0.95	0.58	0.90	1.00	0.81	0.72	0.79	0.87
	QTCSUS	0.35	0.87	0.76	0.89	0.98	1.00	0.98	1.00	1.00	0.91	0.72	1.00	0.95	0.56	0.88	1.00	0.81	0.71	0.67	0.84
	QPN	0.32	0.83	0.75	0.90	0.96	0.91	0.87	0.89	1.00	0.86	0.67	1.00	0.95	0.53	0.86	1.00	0.58	0.68	0.71	0.80
	QPNSUS	0.31	0.82	0.73	0.88	0.96	0.91	0.80	0.87	1.00	0.83	0.66	1.00	0.95	0.52	0.86	1.00	0.56	0.66	0.60	0.79

random-access memory or dynamic random-access memory) of the SSD controller. Therefore, a demand based Q-table cache is more suitable.

Fig. 6.11 shows three types of demand based Q-table cache that we employed in the evaluation. The first type (DE100) uses a Q-table cache with 100 entries and can store up to 500,000 entries in flash memory. When a Q-table cache miss occurs, it evicts some entries from the Q-table cache to flash memory and fetches the necessary information from the latter. Eviction and fetching are performed in units of 20 entries. The second type (DE1000) uses a Q-table cache with 1000 entries and can store up to 500,000 entries in flash memory. Eviction and fetching are performed in units of 100 entries. The last type (DEWO) uses a Q-table cache with 500,000 entries. In this case, we assume that there is no overhead due to eviction or fetching, i.e., the entire Q-table cache with 500,000 entries is placed in DRAM.

Tables 6.23 and 6.24 compare the normalized latency of the our Q-table cache based solution (QTC) and demand based Q-table cache on 3D 512Gb and 3D 128Gb flash memories. In Table 6.23, QTC offers better average latency reductions which are 0.89×(QTC)/1.57×(DE100)/ 1.28×(DE1000)/0.92×(DEQO) on the 3D 512Gb flash memory. The results of the 3D 128Gb flash memory gives average latency reductions which are 0.87×(QTC)/1.82×(DE100)/ 1.44×(DE1000)/0.89×(DEWO). DE100 and DE1000 suffer from considerable eviction and fetching overhead for both flash memory configurations. DE100 has more number of

eviction and fetching operations than DE1000 as the former uses smaller Q-table cache. Therefore, Q-table cache miss occurs more frequently in DE100.

As we mentioned earlier, DEWO has no eviction and fetching overhead. Therefore, for comparing the latency of QTC and DEWO, the observed latency change with the size of QTC can be used. In *TPCC, MSNSFS, RBESQL, DTR*, webmail**, and *RA** workloads, QTC offers better average latency than DEWO. These workloads suffer from the immature state problem that we have already analyzed. In particular, *TPCC* and *webmail** have higher hit rate of 88% and 77% respectively. Increasing the size of QTC could increase the number of immature states, and subsequently worsen the quality of QTC.

• **Q-table cache initialization with average of Q-values:** As we mentioned earlier, Q-table cache has a zero initialization problem. One way to alleviate this is to initialize using the average of the Q-values of the Q-table cache instead of the zero value. Fig. 6.12 shows a Q-table cache initialized with average Q-value. As shown in the figure, Q-value of newly inserted state is initialized with average of Q-values from Q-table cache.

Tables 6.25 and 6.26 compare the normalized latency of the our Q-table cache solution (QTC) and Q-table cache initialized with average Q-value for 3D512G and 3D128G flash memories. QTC offers better average latency reductions which are 0.89×(QTC)/0.94×(AVE) on the

Table 6.23 Latency comparison of demand based Q-table cache for 3D 512Gb flash memory (4CH).

Percentile		TPCC	EXCH24	MSNSFS	MSNSFS*	oltp	RBESQL	DTR*	webmail*	RA*	Financial1	RBESQL*	home1*	web+online*	randtran	rdwhilewr	overwrite	fillunirand	rdrandwrran	V22LUN1	V23LUN2	V24LUN3	AVERAGE
99.9999th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.34	0.92	0.90	1.00	1.00	1.00	0.92	0.86	0.94	1.00	1.00	1.00	0.89	0.99	0.90	0.61	0.93	1.00	0.81	0.76	0.82	0.89
	QPN	0.30	0.82	0.81	1.00	1.00	0.91	0.84	0.80	0.83	0.98	0.96	0.93	0.82	0.99	0.90	0.58	0.86	1.00	0.62	0.65	0.81	0.83
	DE100	1.48	1.23	2.62	3.17	1.00	2.45	2.92	1.10	1.23	1.33	2.53	1.01	0.90	0.99	0.90	1.31	0.93	1.00	1.79	1.46	1.58	1.57
	DE1000	1.20	1.14	1.77	2.18	1.00	1.53	2.54	1.08	1.22	0.98	1.95	1.00	0.86	0.99	0.90	0.94	0.93	1.00	1.31	1.06	1.22	1.28
	DEWO	0.47	0.81	0.95	1.00	1.00	1.07	1.17	0.88	1.04	0.98	1.03	0.99	0.85	0.99	0.90	0.94	0.93	1.00	0.78	0.73	0.79	0.92

Table 6.24 Latency comparison of demand based Q-table cache for 3D 128Gb flash memory (4CH).

Percentile		TPCC	EXCH24	RA*	DTR*	MSNSFS*	RBESQL	MSNSFS	Financial1	oltp	RBESQL*	home2	randtran	rdwhilewr	overwrite	fillunirand	rdrandwrran	V22LUN1	V23LUN2	V24LUN3	AVERAGE
99.9999th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.37	0.93	0.83	0.93	1.00	1.00	0.98	1.00	1.00	0.91	0.75	1.00	0.95	0.58	0.90	1.00	0.81	0.72	0.79	0.87
	QPN	0.32	0.83	0.75	0.90	0.96	0.91	0.87	0.89	1.00	0.86	0.67	1.00	0.95	0.53	0.86	1.00	0.58	0.68	0.71	0.80
	DE100	2.18	1.42	2.66	2.10	3.21	2.65	3.17	0.90	1.00	3.70	1.54	1.00	0.95	1.41	0.91	1.00	1.65	1.47	1.58	1.82
	DE1000	1.76	1.09	2.01	2.07	2.66	1.68	1.89	0.90	1.00	2.50	1.25	1.00	0.95	0.95	0.91	1.00	1.31	1.17	1.25	1.44
	DEWO	0.48	0.83	0.98	1.05	1.03	1.10	0.90	0.88	1.00	1.08	0.77	1.00	0.95	0.95	0.91	1.00	0.76	0.71	0.57	0.89

3D512G flash memory. The results of the 3D 128G flash memory gives average latency reductions which are 0.87×(QTC)/0.89×(AVE).

In particular, in the both flash memories, AVE shows a little latency reduction in the three workloads (*TPCC, rdwhilewr* and *overwrite*) than the other workloads. We analyze that these three workloads give much larger initialization error for newly inserted entries.

Table 6.27 compares the Q-value prediction error of Q-table cache initialized with zero and average Q-value. We define the error for initialization with average Q-value as follows:

$$E_{average} = |Q_{evicted}(s) - Q_{average}(s)| \tag{6.1}$$

where $Q_{evicted}(s)$ and $Q_{average}(s)$ represent Q-value of evicted state s and Q-value of newly inserted state s, respectively.

The error for initialization with zero is defined as follows:

$$E_{zero} = |Q_{evicted}(s) - 0| \tag{6.2}$$

As shown in the Table 6.27, AVE gives relatively larger prediction error than QTC in these three workloads (*TPCC, rdwhilewr, overwrite*).

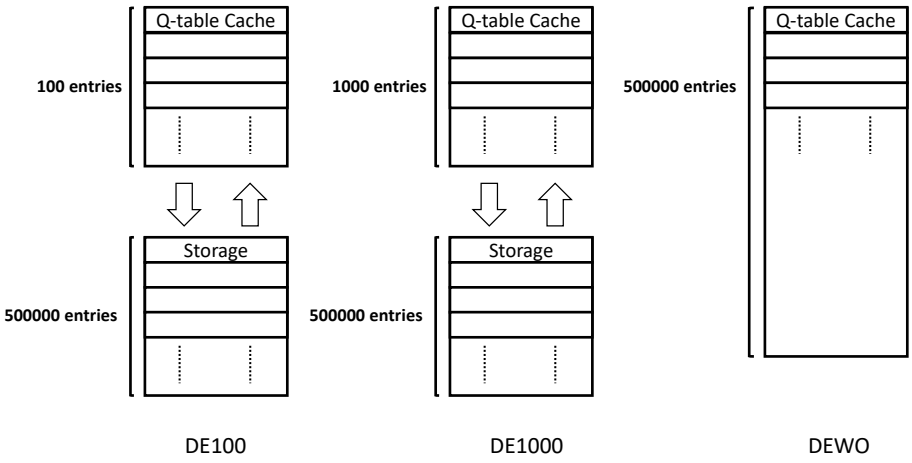


Figure 6.11 Three types of demand based Q-table cache.

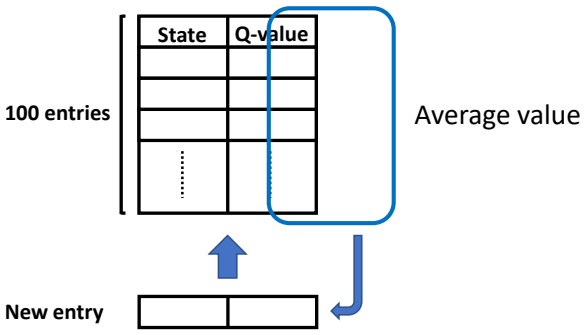


Figure 6.12 Q-table cache initialized with average.

Table 6.25 Latency comparison of Q-table cache initialized with average for 3D 512Gb flash memory (4CH).

Percentile		TPCC	EXCH24	MSNSFS	MSNSFS*	oltp	RBESQL	DTR*	webmail*	RA*	Financial1	RBESQL*	home1*	web+online*	randtran	rdwhilewr	overwrite	fillunirand	rdrandwrran	V22LUN1	V23LUN2	V24LUN3	AVERAGE
99.9999th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.34	0.92	0.90	1.00	1.00	1.00	0.92	0.86	0.94	1.00	1.00	1.00	0.89	0.99	0.90	0.61	0.93	1.00	0.81	0.76	0.82	0.89
	QPN	0.30	0.82	0.81	1.00	1.00	0.91	0.84	0.80	0.83	0.98	0.96	0.93	0.82	0.99	0.90	0.58	0.86	1.00	0.62	0.65	0.81	0.83
	AVE	0.72	0.88	0.88	1.00	1.00	0.96	0.81	0.97	1.01	1.00	1.00	0.93	0.89	0.99	1.42	0.92	1.01	1.00	0.82	0.75	0.81	0.94

Table 6.26 Latency comparison of Q-table cache initialized with average for 3D 128Gb flash memory (4CH).

Percentile		TPCC	EXCH24	RA*	DTR*	MSNSFS*	RBESQL	MSNSFS	Financial1	oltp	RBESQL*	home2	randtran	rdwhilewr	overwrite	fillunirand	rdrandwrran	V22LUN1	V23LUN2	V24LUN3	AVERAGE
99.9999th	Base	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	QTC	0.37	0.93	0.83	0.93	1.00	1.00	0.98	1.00	1.00	0.91	0.75	1.00	0.95	0.58	0.90	1.00	0.81	0.72	0.79	0.87
	QPN	0.32	0.83	0.75	0.90	0.96	0.91	0.87	0.89	1.00	0.86	0.67	1.00	0.95	0.53	0.86	1.00	0.58	0.68	0.71	0.80
	AVE	0.53	0.89	0.92	0.75	1.00	1.03	0.90	1.00	1.00	0.96	0.75	1.00	1.11	0.92	0.99	1.01	0.79	0.71	0.73	0.89

Table 6.27 Q-value prediction error comparison of Q-table cache initialized with average (4CH).

	3D 512Gb				**3D 128Gb**			
	TPCC	DTR*	rdwhilewr	overwrite	TPCC	DTR*	rdwhilewr	overwrite
E_{zero}	0.47	2.44	1.62	1.90	0.22	2.61	0.95	1.00
$E_{average}$	4.60	1.24	3.86	3.35	4.54	1.12	3.90	3.01

Chapter 7

Conculsion and Future Work

7.1 Conclusion

In this dissertation, we addressed the problem of long-tail latency in flash memory-based storage systems. To this end, a reinforcement learning (RL)-assisted GC scheduling technique was proposed. The performance of the three versions of the RL-assisted GC scheduling technique was examined quantitatively.

RL-assisted GC scheduling technique was proposed which learns the storage access behavior online and determines the number of GC operations to exploit the idle time. We also presented aggressive methods, which helps in further reducing the long tail latency by aggressively performing fine-grained GC operations. We evaluated our proposed methods with eight real-world workloads on two types of flash memory storages.

We proposed a technique that dynamically manages key states in RL-assisted GC to reduce the long-tail latency. This technique uses many fine-grained pieces of information as state candidates and manages key

states that suitably represent the characteristics of the workload using a relatively small amount of memory resource. Thus, the proposed method can reduce the long-tail latency even further. In total, eleven workloads are evaluated on two types of flash memory storage.

We also proposed a Q-value prediction network that predicts the initial Q-value of a new state in the Q-table cache. The integrated solution of the Q-table cache and Q-value prediction network can exploit the short-term history of the system with a low-cost Q-table cache. It is also equipped with a small network called Q-value prediction network to make use of the long-term history and provide good Q-value initialization for the Q-table cache. We evaluated the proposed method with 24 workloads on two types of flash memory storage. We also compared our proposed solution with alternative ones including the actor-critic method, Q-value prediction network only and large Q-table cache only methods. We conducted sensitivity analyses to investigate the contribution of each component in our solution, suitable configurations such as the size of the Q-value prediction network, as well as the effects of pre-training the Q-value prediction network. The experiments show that our proposed method reduces by 25%-37% the long tail latency compared to the state-of-the-art method.

7.2 Future Work

Recently, the use of SSD has become widespread, from personal computers to server systems in data centers. The performance of computing units such as CPUs and GPUs is increasing further. The size of data used is also increasing, thereby requiring higher performance of the storage system. In addition, due to the virtualization used in cloud systems, various workload characteristics are mixed and delivered to storage. Therefore, the workload behavior and the requirements for SSDs differ. Based on the study conducted for this dissertation, it is clear that a method that can be easily applied to various SSD specs should be developed.

In this study, we use SSD internal information and workload information as states. However, the RL-assisted GC scheduler can better understand the system behavior using higher layer information such as host. For example, if an SSD can detect changes in access patterns, running programs, or virtual machine behavior, it will be able to learn quickly using large epsilon and learning rates.

In another respect, if the characteristics of the workload are changed, the information learned so far can be saved. If similar workload characteristic occurs and is detected at a later point, the learning starts from the previously stored information. This reduces the probability of selecting an inappropriate action during the learning process and makes learning faster.

Consideration of multi-channel is also an important feature for future work. This study was mainly conducted in 1-channel configuration, and some evaluation was performed in 4-channel configurations. Recent SSDs were shown to actively utilize multi-channels to achieve high capacity and performance. Therefore, it is important to find solutions that perform well in a multi-channel configuration. For example, information related to multi-channels (such as channel blocking state and number of queuing requests) can be added to the state. In addition, various techniques for improving performance in flash memory (or FTL) have been studied and applied to products. It is also important to take advantage of these recent developments in technology.

In this study, we have proposed three solutions based on Q-table as RL based Q-table is simple and can be applied using fewer computational resources. However, this tabular approach has some limitations. For example, the size of the table increases proportionally to the size of the state space; when the state space size grows beyond a certain level, it becomes difficult to apply the approach efficiently. In addition, as binning is required to use a state with continuous values, the performance and resource overhead of RL vary depending on the accuracy of binning. RL techniques using policy approximation have also been studied to overcome the drawbacks of Q-table based methods. These methods use neural networks and can directly use states with consecutive values. It also takes longer to learn and requires more computational overhead than

tabular methods. In order to better understand the environment and determine more appropriate actions, the use of techniques that incorporate policy approximation is required. Due to the nature of the application of SSD, it is difficult to implement policy approximation methods directly. These techniques primarily approximate the policy using iterative learning over a large number of epochs in the same environment. However, in the case of SSD, one of the principal differences is that it operates and learns based on a request received from the host in real time. In this regard, finding and improving RL algorithm that can be effectively applied in SSD is also an important research direction.

Bibliography

[1] Samsung Electronics Co., Ltd., "Samsung v-nand technology," 2014.

[2] C. Kim, J. Cho, W. Jeong, I. Park, H. Park, D. Kim, D. Kang, S. Lee, J. Lee, W. Kim, J. Park, Y. Ahn, J. Lee, J. Lee, S. Kim, H. Yoon, J. Yu, N. Choi, Y. Kwon, N. Kim, H. Jang, J. Park, S. Song, Y. Park, J. Bang, S. Hong, B. Jeong, H. Kim, C. Lee, Y. Min, I. Lee, I. Kim, S. Kim, D. Yoon, K. Kim, Y. Choi, M. Kim, H. Kim, P. Kwak, J. Ihm, D. Byeon, J. Lee, K. Park, and K. Kyung, "11.4 a 512gb 3b/cell 64-stacked wl 3d v-nand flash memory," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 202–203, Feb 2017.

[3] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, (New York, NY, USA), pp. 229–240, ACM, 2009.

[4] S. Choi, D. Kim, S. Choi, B. Kim, S. Jung, K. Chun, N. Kim, W. Lee, T. Shin, H. Jin, H. Cho, S. Ahn, Y. Hong, I. Yang, B. Kim, P. Yoo, Y. Jung, J. Lee, J. Shin, T. Kim, K. Park, and J. Kim, "19.2 a 93.4mm2 64gb mlc nand-flash memory with 16nm cmos technology," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 328–329, Feb 2014.

[5] C. Kim, D. Kim, W. Jeong, H. Kim, I. H. Park, H. Park, J. Lee, J. Park, Y. Ahn, J. Y. Lee, S. Kim, H. Yoon, J. D. Yu, N. Choi, N. Kim, H. Jang, J. Park, S. Song, Y. Park, J. Bang, S. Hong, Y. Choi, M. Kim, H. Kim, P. Kwak, J. Ihm, D. S. Byeon, J. Lee, K. Park, and K. Kyung, "A 512-gb 3-b/cell 64-stacked wl 3-d-nand flash memory," *IEEE Journal of Solid-State Circuits*, vol. 53, pp. 124–133, Jan 2018.

[6] R. Micheloni, S. Aritome, and L. Crippa, "Array architectures for 3-d nand flash memories," *Proceedings of the IEEE*, vol. 105, pp. 1634–1649, Sep. 2017.

[7] C. Monzio Compagnoni, A. Goda, A. S. Spinelli, P. Feeley, A. L. Lacaita, and A. Visconti, "Reviewing the evolution of the nand flash technology," *Proceedings of the IEEE*, vol. 105, pp. 1609–1633, Sep. 2017.

[8] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, pp. 74–80, Feb. 2013.

[9] W. Kang, D. Shin, and S. Yoo, "Reinforcement learning-assisted garbage collection to mitigate long-tail latency in ssd," *ACM Trans. Embed. Comput. Syst.*, vol. 16, pp. 134:1–134:20, Sept. 2017.

[10] W. Kang and S. Yoo, "Dynamic management of key states for reinforcement learning-assisted garbage collection to reduce long tail latency in ssd," in *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, (New York, NY, USA), pp. 8:1–8:6, ACM, 2018.

[11] W. Kang and S. Yoo, "Q-value prediction for reinforcement learning assisted garbage collection to reduce long tail latency in ssd," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019. Early Access.

[12] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The tail at store: A revelation from millions of hours of disk and {SSD} deployments," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pp. 263–276, 2016.

[13] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. 2016.

[14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[15] Q. Zhang, X. Li, L. Wang, T. Zhang, Y. Wang, and Z. Shao, "Lazy-rtgc: A real-time lazy garbage collection mechanism with jointly optimizing average and worst performance for nand flash memory storage systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, pp. 43:1–43:32, June 2015.

[16] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, "Real-time garbage collection for flash-memory storage systems of real-time embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 3, pp. 837–863, Nov. 2004.

[17] S. Choudhuri and T. Givargis, "Deterministic service guarantees for nand flash using partial block cleaning," in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '08, (New York, NY, USA), pp. 19–24, ACM, 2008.

[18] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "Real-time flash translation layer for nand flash memory storage systems," in *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pp. 35–44, April 2012.

[19] N. Shahidi and M. T. Kandemir, "Cachedgc: Cache-assisted garbage collection in modern solid state drives," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 79–86, Sept 2018.

[20] Qingsong Wei, Bozhao Gong, S. Pathak, B. Veeravalli, LingFang Zeng, and K. Okada, "Waftl: A workload adaptive flash translation layer with data partition," in *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, May 2011.

[21] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds," *ACM Trans. Storage*, vol. 13, pp. 22:1–22:26, Oct. 2017.

[22] G. Amvrosiadis, A. D. Brown, and A. Goel, "Opportunistic storage maintenance," in *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, (New York, NY, USA), pp. 457–473, ACM, 2015.

[23] J. He, D. Nguyen, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Reducing file system tail latencies with chopper," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, (Santa Clara, CA), pp. 119–133, USENIX Association, Feb. 2015.

[24] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Split-level i/o scheduling," in *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, (New York, NY, USA), pp. 474–489, ACM, 2015.

[25] L. Han, Y. Ryu, and K. Yim, "Cata: A garbage collection scheme for flash memory file systems," in *Ubiquitous Intelligence and Computing* (J. Ma, H. Jin, L. T. Yang, and J. J.-P. Tsai, eds.), (Berlin, Heidelberg), pp. 103–112, Springer Berlin Heidelberg, 2006.

[26] M. Lin and S. Chen, "Efficient and intelligent garbage collection policy for nand flash-based consumer electronics," *IEEE Transactions on Consumer Electronics*, vol. 59, pp. 538–543, August 2013.

[27] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 39–50, IEEE Computer Society, 2008.

[28] A. Pritzel, B. Uria, S. Srinivasan, A. P. Badia, O. Vinyals, D. Hassabis, D. Wierstra, and C. Blundell, "Neural episodic control," in *Proceedings of the 34th International Conference on Machine Learning* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings*

of Machine Learning Research, (International Convention Centre, Sydney, Australia), pp. 2827–2836, PMLR, 06–11 Aug 2017.

[29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[30] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems 12* (S. A. Solla, T. K. Leen, and K. Müller, eds.), pp. 1008–1014, MIT Press, 2000.

[31] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," in *2009 First International Conference on Advances in System Simulation*, pp. 125–131, Sept 2009.

[32] SNIA, "I/o trace data files," 2008.

[33] Filebench, "filebench/filebench," 2016.

[34] M. Bilal and S.-G. Kang, "A cache management scheme for efficient content eviction and replication in cache networks," *IEEE Access*, vol. 5, pp. 1692–1701, 2017.

[35] G. Bebis and M. Georgiopoulos, "Feed-forward neural networks," *IEEE Potentials*, vol. 13, pp. 27–31, Oct 1994.

[36] R. HECHT-NIELSEN, "Iii.3 - theory of the backpropagation neural network," in *Neural Networks for Perception* (H. Wechsler, ed.), pp. 65 – 93, Academic Press, 1992.

[37] J. Heaton, *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[38] M. Kwon, J. Zhang, G. Park, W. Choi, D. Donofrio, J. Shalf, M. Kandemir, and M. Jung, "Tracetracker: Hardware/software co-evaluation for large-scale i/o workload reconstruction," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 87–96, Oct 2017.

[39] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, (Piscataway, NJ, USA), pp. 61:1–61:12, IEEE Press, 2016.

[40] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Accelerating dependent cache misses with an enhanced memory controller," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 444–455, June 2016.

[41] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory bandwidth management for efficient performance isolation in

multi-core platforms," *IEEE Transactions on Computers*, vol. 65, pp. 562–576, Feb 2016.

[42] W. Wang, J. W. Davidson, and M. L. Soffa, "Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 419–431, March 2016.

[43] S. Jiang and X. Zhang, "Making lru friendly to weak locality workloads: a novel replacement algorithm to improve buffer cache performance," *IEEE Transactions on Computers*, vol. 54, pp. 939–952, Aug 2005.

[44] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache.," in *FAST*, vol. 3, pp. 115–130, 2003.

[45] AVNET, "Zedboard technical specifications," 2017.

[46] ARM, "Cortex-r – arm developer," 2016.

[47] Marvell, "Marvell 88nv11xx ssd controllers," 2017.

[48] Marvell, "Marvell nvme pcie gen3x4 ssd controllers," 2018.

[49] Y. Kang, Y. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart ssd," in *2013 IEEE 29th Symposium on*

Mass Storage Systems and Technologies (MSST), pp. 1–12, May 2013.

[50] Samsung, "Smartssd - samsung @firstsamsung."

[51] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: Opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 1221–1230, ACM, 2013.

[52] B. Tallis, "The samsung 970 evo plus (250gb, 1tb) nvme ssd review: 96-layer 3d nand," Jan 2019.

[53] B. Tallis, "The samsung 860 qvo (1tb, 4tb) ssd review: First consumer sata qlc," Nov 2018.

[54] B. Tallis, "The crucial p1 1tb ssd review: The other consumer qlc ssd," Nov 2018.

[55] "A persistent key-value store."

[56] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara, "Understanding storage traffic characteristics on enterprise virtual desktop infrastructure," in *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, (New York, NY, USA), pp. 13:1–13:11, ACM, 2017.

[57] G. Wu and X. He, "Reducing ssd read latency via nand flash program and erase suspension.," in *FAST*, vol. 2, p. 3, 2012.

국문초록

낸드 플래시 메모리는 실시간 임베디드 시스템으로부터 고성능의 엔터프라이즈 서버 시스템까지 다양한 시스템에서 널리 사용 되고 있다. 플래시 메모리는 (1) erase-before-write (write once)와 (2) endurance 문제를 갖고 있다. Erase-before-write 특성을 다루기 위해 flash-translation layer (FTL)을 적용 한다. 현재 플래시 메모리의 write-once 특성과 block erase특성으로 인한 latency 증가를 감소 시키기 위하여 page-level mapping방식이 주로 사용 된다.

Garbage collection (GC)은 99^{th} percentile에서 평균 지연시간의 100배 이상 증가하는 long tail latency를 유발시키는 주요 원인 중 하나이다. 따라서 실시간 시스템이나 quality-critical system에서는 Quality of Service (QoS) 제한과 같은 주어진 요구 조건을 만족 시킬 수 없다.

플래시 메모리의 용량이 증가함에 따라 GC latency도 증가하는 경향을 보인다. 이것은 플래시 메모리의 용량이 증가 함에 따라 플래시 메모리의 블록 크기 (하나의 블록이 포함하고 있는 페이지의 수)가 증가하기 때문이다. GC latency는 valid page copy와 block erase 시간에 의해 결정 된다. 따라서, 블록 크기가 증가하면, GC latency도 증가 한다.

특히, 최근 2D planner 플래시 메모리에서 3D vertical 플래시 메모리 구조로 전환됨에 따라 블록 크기는 증가 하였다. 심지어 3D vertical 플래시 메모리에서도 블록 크기가 지속적으로 증가 하고 있다. 따라

서 3D vertical 플래시 메모리에서 long tail latency 문제는 더욱 심각해진다.

본 논문에서 우리는 강화학습(Reinforcement learning, RL)을 이용한 세 가지 버전의 새로운 GC scheduling 기법을 제안하였다. 제안된 기술의 목적은 스토리지 시스템의 idle 시간을 활용하여 GC에 의해 발생된 long tail latency를 감소 시키는 것이다. 또한, 우리는 RL-assisted GC 솔루션을 위한 정량 분석 하였다.

우리는 스토리지의 access behavior를 온라인으로 학습하고, idle 시간을 활용할 수 있는 GC operation의 수를 결정하는 RL-assisted GC scheduling 기술을 제안 하였다. 추가적으로 우리는 공격적인 방법을 제시 하였다. 이 방법은 작은 단위의 GC operation들을 공격적으로 수행 함으로써, long tail latency를 더욱 감소 시킬 수 있도록 도움을 준다.

또한 우리는 long tail latency를 더욱 감소시키기 위하여 RL-assisted GC의 key state들을 동적으로 관리할 수 있는 Q-table cache 기술을 제안 하였다. 이 기술은 state 후보로 매우 많은 수의 세밀한 정보들을 사용하고, 상대적으로 작은 메모리 공간을 이용하여 workload의 특성을 적절하게 표현 할 수 있는 key state들을 관리 한다. 따라서, 제안된 방법은 long tail latency를 더욱 감소 시킬 수 있다.

추가적으로, 우리는 Q-table cache에 새롭게 추가되는 state의 초기 값을 예측하는 Q-value prediction network (QP Net)를 제안 하였다. Q-table cache와 QP Net의 통합 솔루션은 저 비용의 Q-table cache를 이용하여 단기간의 과거 정보를 활용 할 수 있다. 또한 이것은 QP Net이라고 부르는 작은 신경망을 이용하여 학습한 장기간의 과거 정보를

사용하여 Q-table cache에 새롭게 삽입되는 state에 대해 좋은 Q-value 초기값을 제공한다. 실험결과는 제안한 방법이 state-of-the-art 방법에 비교하여 25%-37%의 long tail latency를 감소 시켰음을 보여준다.

주요어: 긴 꼬리 지연시간, 강화학습, 가비지 컬렉션, 솔리드 스테이트 드라이브, 낸드 플래시 메모리, 스토리지

학번: 2016-30281